AFIT/GCS/ENG/92D-01

CREATING AND MANIPULATING FORMALIZED SOFTWARE ARCHITECTURES TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

THESIS

Cynthia Griffin Anderson
Captain, USAF

AFIT/GCS/ENG/92D-01

Approved for public release; distribution unlimited

AFIT/GCS/ENG/92D-01

CREATING AND MANIPULATING FORMALIZED SOFTWARE ARCHITECTURES TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Cynthia Griffin Anderson, B.S.C.S.

Captain, USAF

December, 1992

Preface

This research was part of an effort to develop a technology which will allow sophisticated end-users to formally specify and compose software applications using domain-oriented, rather than programming-oriented, terms. Ultimately, such a technology will enable users to compose applications to suit their requirements, to execute prototypes of these composed applications to verify that they behave as expected/desired and, then, to automatically generate efficient software code to satisfy the original requirements. This thesis investigated the role of software architectures in the development of such an application composition system which has been named Architect. I hope that the Architect system implemented herein will prove to be a useful starting point in achieving the overall research goal of developing a full-scale application generation system.

I wish to thank my committee members, Majors Gregg Gunsch and Dave Luginbuhl, for their incisive comments on the draft of this document. A special thanks goes to my thesis advisor, Major Paul Bailor, whose wisdom, advice, encouragement and confidence in me made this thesis possible. I also wish to thank the other members of the formal methods research group, especially Mary Anne Randour who eagerly shared her wealth of REFINE expertise with all of us.

But most of all, I want to thank my husband and best friend, Andy, for his patience, support, understanding, and encouragement throughout this AFIT program and in all my endeavors.

Cynthia Griffin Anderson

Table of Contents

List of Figures

List of Tables

Abstract

This research investigated technology which enables sophisticated users to specify, generate, and maintain application software in domain-oriented terms. To realize this new technology, a development environment, called Architect, was designed and implemented. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. Architect depends upon the existence of a formal object base (or domain model) which was investigated by another student in related research. The research described in this thesis relied on the concept of a software architecture, which was a key to Architect's successful implementation. Various software architectures were evaluated and the Object-Connection-Update (OCU) model, developed by the Software Engineering Institute, was selected. The Software Refinery environment was used to implement the composition process which encompasses connecting specified domain objects into a composed application, performing semantic analysis on the composed application, and, if no errors are discovered, simulating the execution of the application. Architect was validated using both artificial and realistic domains and was found to be a solid foundation upon which to build a full-scale application composition system.

CREATING AND MANIPULATING FORMALIZED SOFTWARE ARCHITECTURES TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

I. Introduction

1.1 Background

A "software crisis" is upon us, characterized by expensive, often late, often unreliable and difficult to maintain systems which seldom meet all their requirements (6:7–8). As computer hardware becomes more powerful and significantly less expensive, it becomes possible to find automated solutions to more and more problems (6:8). These formerly marginal application areas often require very large, very complex software systems and "software entities are more complex for their size than perhaps any other human construct;... many of the classical problems of developing software products derive from this essential complexity and its non-linear increases with size" (9:11). As if this weren't bad enough, trends indicate a widening gap between the productivity of an insufficient number of computer professionals and the demand for their services, as illustrated in Figure 1.1 (6:10). Clearly, something must be done to improve quality and increase productivity in the software development process.

One technique touted to achieve these quality and productivity improvements is software reuse. In terms of the software development process, "reuse is very simply any procedure that produces (or helps produce) a system by reusing something from a previous development effort" (15:2). But to obtain its maximum benefits, software reuse should be a "process of reusing software that was designed to be reused" (3:ix). This distinction is important as it implies a more systematic, formal approach whose *primary* objective is to create reusable software components, not develop them as an accidental by-product. Systematic reuse, most notably design reuse, offers the promise of decreasing overall soft-

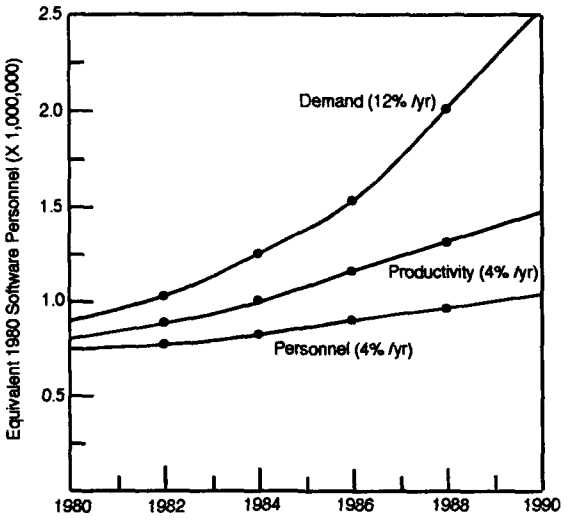

Figure 1.1. Software Development Trends

ware development costs, improving maintainability, increasing understandability, reducing complexity and improving reliability (17).

Traditional engineering disciplines have long recognized and successfully employed reuse of design products. They benefit from large bodies of scientific knowledge, which over the years, have been codified into models (12). These models act as reusable templates from which to construct practical, working solutions to problems in a specific engineering area. For example, "automotive engineers have models of cars, civil engineers have models of bridges, mechanical engineers have models of rolling mills, electrical engineers have models of motors..." (12:140). The engineer is trained to understand these models, to recognize which model will solve a particular problem, and to adapt the model, if necessary, to fit a specific application. He also knows that proper usage of the appropriate model *will* produce the desired result even before he attempts to build the actual object.

Software engineering is a relatively new discipline which is just now beginning to codify its body of knowledge. Unlike other engineering disciplines, it does not currently rely on models as a means of designing working solutions. Instead, each new problem is treated in isolation, as a unique situation requiring a completely unique solution. D'Ippolito maintains that "models can do for the software industry what they have done before and

continue to do for the main-line engineering professions" (11:258). That is, they can provide "reuse at the design level, reduced system complexity, a means to measure project risk, reduced coding costs, reduced testing costs, reduced documentation costs, and increased maintainability and enhanceability" (11:258).

We can think of a model as an architecture or a blueprint for building something. When computer scientists discuss computer hardware, they implicitly or explicitly reference its structural composition, its schematic diagram, its architecture. The concept of an architecture for computer hardware is clearly understood in the computer science community; the term is now being applied to computer software as well. A software architecture is "the high-level packaging structure of functions and data, their interfaces and control, to support the implementation of applications in a domain" (20:3). More simply, it describes the components which constitute a software system and how those components are connected (a more precise definition of the term "component" will be presented in Chapter 3). If the needed components already existed in a library and if the appropriate components could be easily identified, retrieved, and connected, reliable software systems could be designed and implemented very quickly.

The Software Architectures Engineering (SAE) Project at the Software Engineering Institute, an affiliate of Carnegie-Mellon University, is researching the feasibility of such a process, which they have named "model-based software development" (24). In this context, a model may be thought of as a set of software components or modules, each performing a well-defined operation or function. In traditional engineering disciplines, the models and the rules for combining them are stored in public technology bases where they are readily available to anyone who wants to use them (24:7). The SAE project seeks to develop such a technology or knowledge base for various software application areas and domains. Developers of new software systems in these specific domains will be able to choose appropriate components from the knowledge base and combine those components using appropriate connection rules (also in the knowledge base) to create a software architecture which represents the desired software system.

1.2 Problem

As in the traditional engineering disciplines (where, for example, automotive engineers design automobiles, not bridges or airplanes), model-based software development is likely to be conducted within a particular domain or application area. To make it a reality, there must be a set of readily-available software components, a way for the software developer to quickly and easily select the components required to construct his particular application, and a means to compose those components in a meaningful way to produce the desired application. An obvious approach to this problem would be to develop an automated system to assist in this process. To do so,

- there must be a formal description (both in human- and machine-understandable terms) of the available components,
- there must be a formal definition of the software architecture or framework into which the components can be placed, and
- there must be a method by which the software developer can specify the application that he wants to create.

Figure 1.2 provides a simplified illustration of a domain-specific application composition system. A domain analysis, conducted by experts in the application area, provides the basis for identifying and constructing a set of appropriate domain-specific components. A generalized software architecture provides the basis for developing an architecture customized for the domain. A domain-specific language allows the software developer to specify, in domain-oriented terms, the desired application which is constructed using the domain-specific architecture and appropriate domain components from the technology base.

1.2.1 Problem Statement

Develop a formalized model of a software architecture and implement it within a domain-specific application composition system.

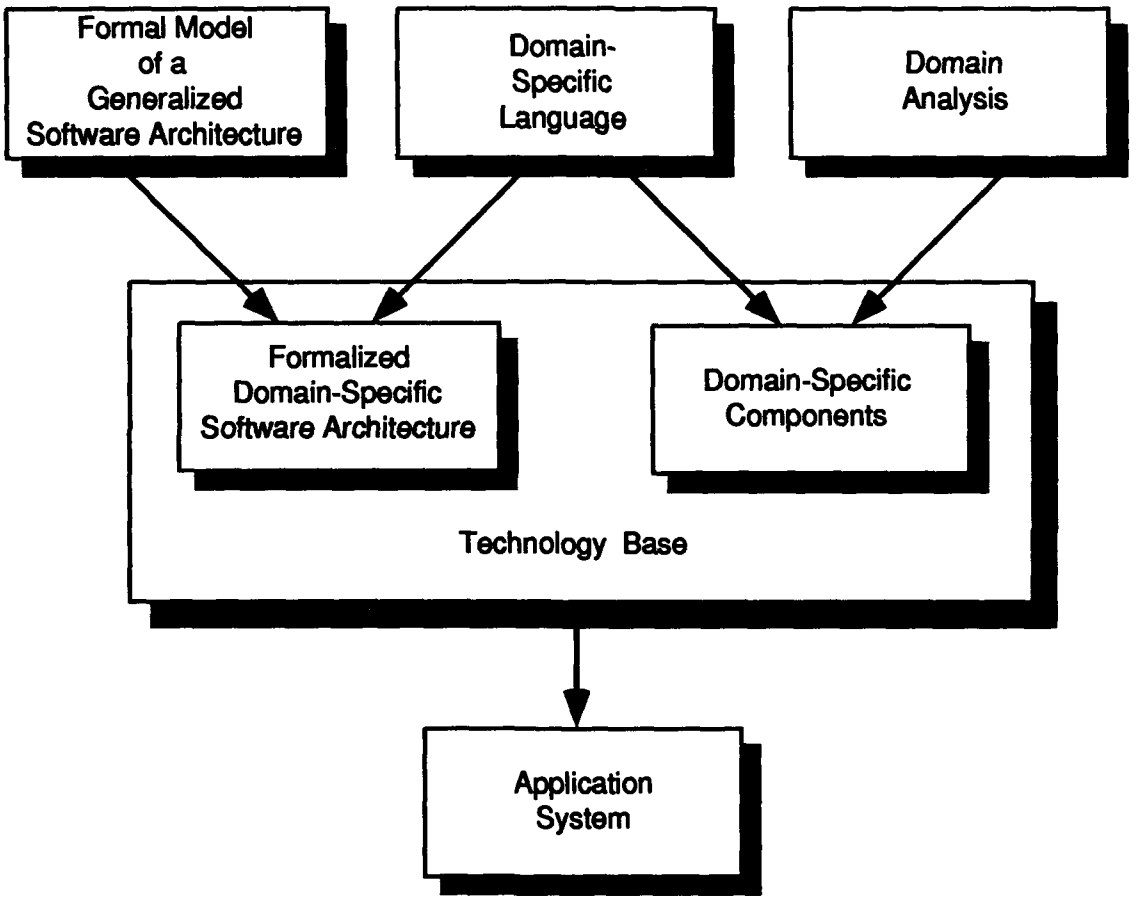

Figure 1.2. Domain-Specific Software Application Composition Methodology

1.3 Scope

This research effort focuses on formalizing an appropriate software architecture and implementing it within a domain-specific application composition system. The implementation will enforce the rules of composition established by the software architecture model and simulate the behavior of the composed application, using formal specification technology. This enables the developer to verify that the system behaves as intended *before* it is actually coded (with automated assistance) in an implementation language such as Ada.

There are many additional elements of a useful and usable application composition system; several related research efforts are currently underway at the Air Force Institute of Technology to address them. Most notably, Captain Mary Anne Randour has developed a language with which the software developer can formally specify his application in domain-oriented (rather than programming-oriented) terms (33) and Lieutenant Timothy Weide is developing a visual interface to facilitate application specification and composition (44).

1.4 Sequence of Presentation

The remainder of this thesis is organized as follows:

Chapter II provides a review of the available literature concerning software architectures.

Chapter III describes the application composition system of which this thesis effort is a part. It serves as a requirements analysis.

Chapter IV presents an overview of the design of the application composition system introduced in Chapter III and discusses the specific software architectural model which was used as its foundation.

Chapter V explains the detailed design of the application composer implemented during this research effort.

Chapter VI demonstrates how the application composer was used to compose meaningful applications within a specific domain.

Chapter VII contains conclusions about the work described herein and presents recommendations for further research.

Several appendices provide additional information for the interested reader.

Appendix A serves as a primer for formally describing architectural components.

Appendix B summarizes detailed, background knowledge about executing the application composer which may be needed by follow-on researchers as they strive to extend the system.

Appendix C displays sample composed applications for the domain discussed in Chapter VI.

Appendix D contains the application composer's domain-independent source code.

Appendix E contains the domain-specific technology base for the system's validating domain.

II. Survey of Current Literature

2.1 Introduction

"Architecture" is a common and well-understood term when applied to computer hardware. Even people who are newly acquainted with computer science have assimilated the concept of a computer architecture with a mental image of boxes representing the CPU, main memory, I/O devices, etc., connected by data and address buses. When computer scientists discuss computer hardware, they implicitly or explicitly reference its structural composition, its schematic diagram, its architecture. To them, the term "architecture" is synonymous with structure, organization, and how components are connected together. Now the term is being used to describe software as well.

This chapter surveys software architectures in the literature. What are they? How are they developed? How have they been successfully used? This survey is not limited to a particular time period, although serious interest in the topic appears to have begun in the mid 1980s.

2.2 Description of Software Architectures

The American Heritage Dictionary defines architecture as "a style and method of design and construction." In Model-Based Software Development, an architecture is "a selection, from a technology base, of models and composition rules that defines the structure, performance, and use of a system relative to a set of engineering goals" (24:8), where a model is simply "reusable engineering experience" (24:2). Kang defines a software architecture as "the high-level packaging structure of functions and data, their interfaces and control, to support the implementation of applications in a domain" (20:3) and a domain as "a set of current and future applications which share a set of common capabilities and data" (20:3). As referenced by Lane, Shaw expands the concept of a software architecture with this definition: "the study of large scale structure and performance of software systems" (22:1). Clearly, all these definitions have "structure" in common; in its most general form, then, a software architecture somehow represents the structure of a software system.

It is human nature to be confused and uncomfortable with complexity. Developing ever larger and more complex software systems leads to problems describing the system-level design; i.e., the kinds of modules used in the system and how those modules are connected. This system-level design or software architecture level "requires new kinds of abstractions that capture essential properties of the major subsystems and the ways they interact" (38:143). Abstraction is a process which allows us to reduce or manage complexity, extracting only essential elements or qualities from the actual physical object or concept and ignoring non-essential details. Software architectures provide a means for describing these abstractions.

There are many similarities in the way existing software systems are organized or structured. These common architectures can be grouped into the following broad categories (38:143-4):

1. Pipes and Filters: Each module receives inputs and transforms those inputs in some meaningful way into outputs, which then become the inputs of another module. Modules are connected when the output of one module serves as the input to another.

Figure 2.1. Filters and Pipes

2. Data Abstraction: Each module represents an object and its associated operations. The connections in this type of architecture represent one object invoking another object's operation. This approach is also known as object-oriented design.

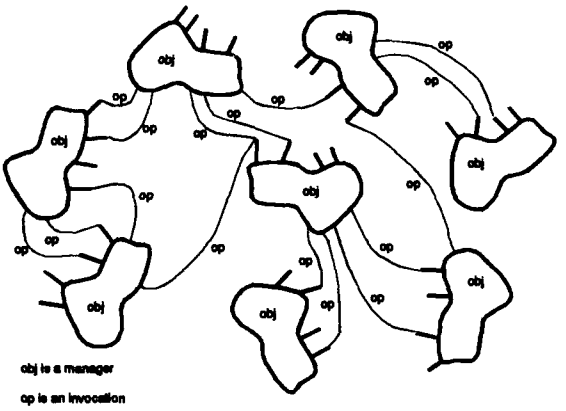

Figure 2.2. Data Abstraction

3. Layered Systems: The system is organized hierarchically, with each layer providing services to the layer above, while receiving services from the layer below. This is a common architecture for operating systems.

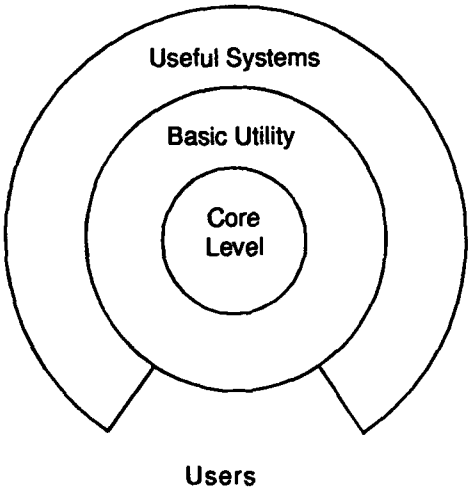

Figure 2.3. Layered Systems

4. Rule-Based Systems: A computational mechanism sequentially applies a collection of applicable rules from a knowledge base. Each rule specifies the condition under which it can be executed and the action that will be taken when it does execute.

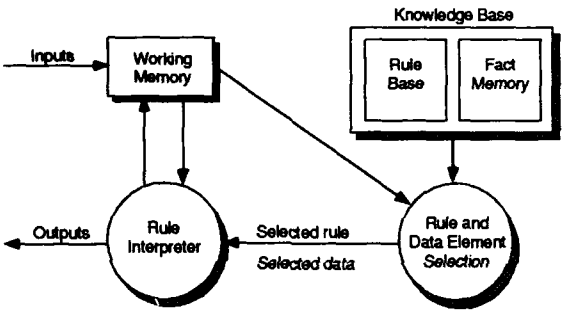

Figure 2.4. Rule-Based Systems

5. Blackboard Systems: A central data structure (representing the state of the computation) is surrounded by independent processes which check its status and execute if they can further the calculation and/or enable another process to execute.

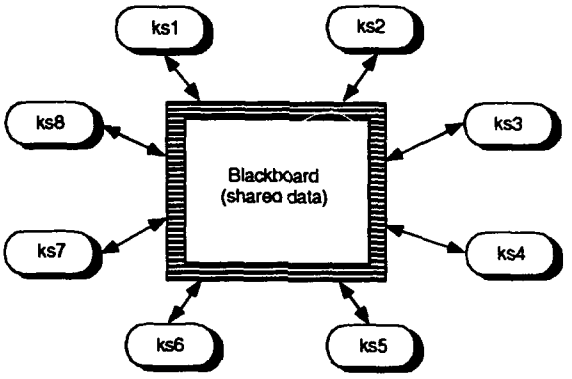

Figure 2.5. Blackboard Systems

Although these basic architectures provide meaningful abstractions to help us grasp how systems are structured and can be used to describe a wide variety of software systems, they are currently used only informally, are not widely understood, and are not systematically taught to computer professionals (38:144). This is unfortunate because architectural analysis can support reuse of software development products by focusing attention on the

high-level design or system framework and by increasing understanding of the relationship between system organization and system behavior (29:125). To obtain the maximum benefits offered by software architectures, certain information about components and connections must be available. Such information includes informal descriptions, abstract models, syntax, semantics, evaluation criteria, engineering considerations, etc. (38:145). Difficulties in identifying, codifying and disseminating this architectural information have kept software architectures from being used more extensively (38:145).

2.3 Developing Software Architectures

In 1988, Shaw wrote: "identifying and classifying system functions that are common to many applications is a significant first step to the development of software architectures" (38:145). However, by 1990, she had come to agree with other researchers (1, 17, 43) that domain-specific architectures would likely lead to more reuse and now feels that we must start by identifying and classifying system functions that are common to a particular domain (29:123).

One approach to identifying system functions is domain analysis. Arango quotes Neighbors: "a domain analysis is an attempt to identify the objects, operations and relationships between what domain experts perceive to be important about the domain" (1:153). Arango divides domain analysis into two phases (1:153):

1. Conceptual analysis - the identification and acquisition of information needed to specify the system.
2. Constructive analysis - the identification and acquisition of information needed to implement the system.

Because most domains are quite stable, Arango advocates using "practical domain analysis methods" which incrementally augment or refine existing domain knowledge based on studies by domain experts and analysis of documentation from similar existing systems (1:159).

Feature-oriented domain analysis (FODA) is one approach to domain analysis whose primary goal is to make domain products reusable (20:47). A domain model describes

the problems within a domain which can be solved with software systems; it defines the problem space and is analogous to Arango's conceptual analysis phase. A domain model is "a definition of the functions, objects, data and relationships of a domain" (20:3). The domain modeling component of FODA consists of three activites: feature analysis, entity-relationship modeling and functional analysis (20:35). Feature analysis identifies the features (user-visible characteristics) of the domain, then abstracts and formally describes them. The entity-relationship model captures and defines domain knowledge by identifying entities and their relationships and classifying these entities into homogenous sets (20:41). Presumably, features map to entities in some way. Finally, the functional analysis identifies commonalities and differences between applications within the domain (20:42).

A second aspect of FODA is the architectural model. It provides a software solution to the problems defined in the domain modeling phase (20:47); it defines the solution space and is analogous to Arango's constructive analysis phase. Architecture modeling concentrates on identifying the processes and domain-specific modules required to satisfy the solution and allocating the features, functions, and data objects defined in the domain model to these processes and modules (20:47). For maximum flexibility and adaptation to future changes, a layered architecture is used as it allows the system to be viewed from various levels of abstraction and encourages reuse at the appropriate level for each application. More productivity is achieved through reuse at the higher design levels (20:50).

Classifying system functions or architectural components presents a major challenge. To be effective, any classification scheme must be consistent (the same item is classified the same way every time), expressive (able to communicate all required information), and understandable (14:303). Classification methods, which are heavily oriented toward indexing into very large libraries include (14):

1. enumerated classification - the domain is divided into successively narrower classes in a rigid tree-structured hierarchy.
2. faceted classification - the domain is divided into its elemental classes or facets. Components in the domain are described by combining these basic classes in a more flexible structure than is possible with the enumerated scheme.

2.4 *Examples of Software Architecture Use*

There are several examples of the successful application of software architectures.

2.4.1 US Army Information Systems Engineering Command Softech, Inc analyzed seven typical Combat Service Support systems for the US Army to evaluate whether any functions were common to multiple applications and discovered that every application included an inventory management function (37:16). They constructed a generic architecture for that function and coded it using Ada language packages, tailoring the packages to each application, as necessary. Although the applications differed significantly (i.e., personnel, logistics, etc.), the generic architecture allowed the inventory function to be treated as a single entity and to be reused across seven application areas. This study demonstrated the feasibility of reusing software components and, to some extent, higher-level designs across several different application areas (37).

2.4.2 Object-Oriented Design (OOD) Paradigm The Software Engineering Institute, while working on the Ada Simulator Validation Program, has developed a model or architecture for a flight simulator (23). Each real-world component of an airplane (engine, electrical system, fuel system, etc.) is represented as an object and encapsulated as a system. Communication among the systems is accomplished via connection modules which provide the only interface between the systems and their environment. See Figure 2.6. The architecture provides a means to systematically specify objects, systems, and their connections and encourages consistent implementation (23:41), which results in better understandability and improved maintainability. Although originally proposed as a model for a flight simulator only , this architecture has been used to represent an elevator system (40), a cruise control system (40), and an electrical system (7), among others.

2.4.3 Feature-Oriented Domain Analysis (FODA) Kang and others used the complete FODA methodology to successfully develop a window management system (20).

2.4.4 Hierarchical Software Systems. A layered or hierarchical architecture is the foundation of research conducted at the University of Texas. Analysis of two unrelated

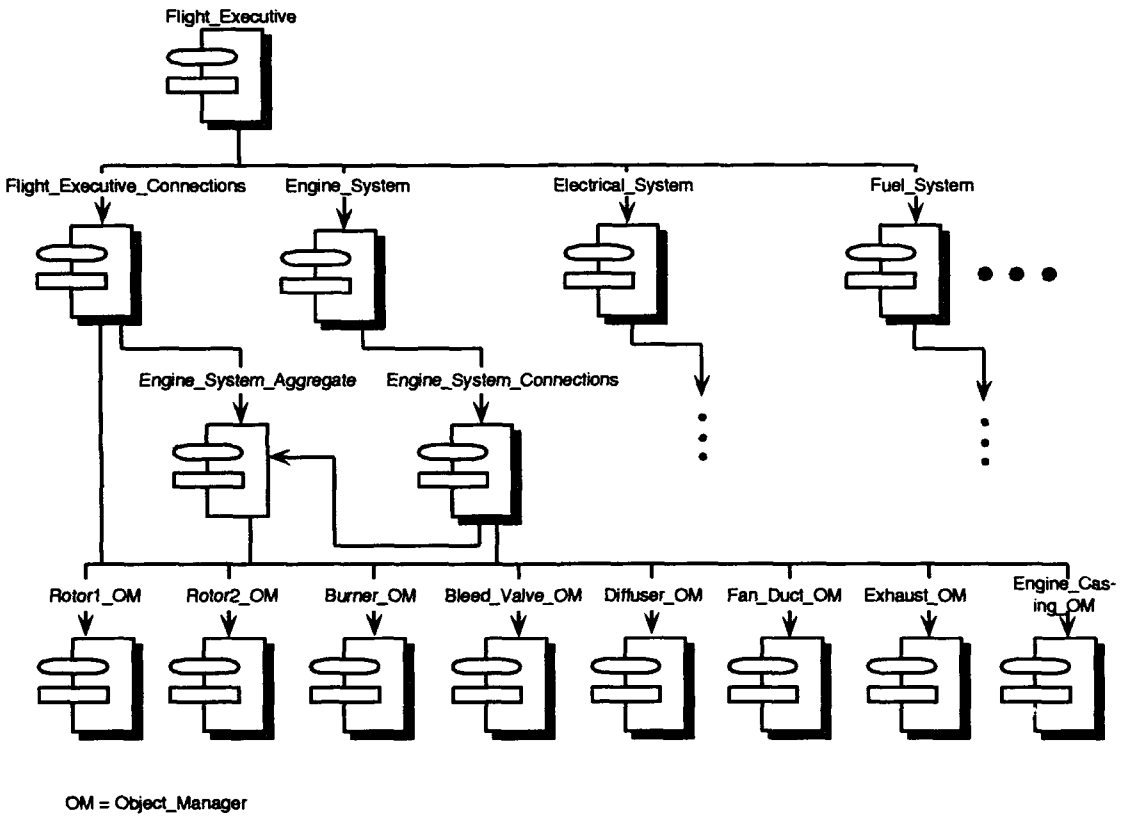

Figure 2.6. OOD Paradigm for a Flight Simulator

projects (GENESIS and Avoca) revealed striking similarities in design and organization (5). Based on those similarities, a model was developed which uses sets of "plug-compatible and interchangeable" (5:3) components and provides a straight-forward means to indicate how the components are connected to create a system (via composition rules which provide the "guidelines by which components can be glued together" (5:4)). This approach encourages component reuse and provides a standardized design process which can be used to create systems very quickly; one database management system was designed, composed and implemented within 20 minutes (5:2). Batory and O'Malley assert that hierarchical designs can be used for a wide range of application areas (even some real-time, performance-driven applications where other layered designs have resulted in slow implementations (5:36)); as an example, their method is currently being used to design an upgrade to the Mach operating system (5:40).

2.4.5 Flight Dynamics Division (FDD) of the Goddard Space Flight Center The FDD has been very successful using a modified version of the General Object-Oriented Design (GOOD) methodology to develop various simulators. They rely heavily on three concepts from GOOD: abstraction, inheritance and domain-specific architectures (41:278). The Upper Atmosphere Research Satellite Telemetry Simulator (UARSTELS) benefitted from several lessons learned from past development efforts. Instead of using a highly nested architecture which was found to greatly increase compilation overhead, a non-nested architecture of Ada generic packages was used (41:282). The Generic Dynamics and Telemetry Simulator (GENSIM) used an object-oriented design and an architecture very similar to the SEI Flight Simulator (minus the connection modules). Significant lessons learned by FDD from these projects include the fact that high compilation overhead is caused by highly nested architectures, the advantage of using an object-oriented design approach, and the importance of building domain components before developing an architecture (according to the authors, this provides more potential for reuse across multiple architectures) (41).

2.4.6 Command Center Processing and Display System Replacement (CCPDS-R) TRW used a Software Architecture Skeleton (SAS) to provide a software structure which "identifies all top-level executable components, all control interfaces between these compo-

nents and all type definitions and data interfaces between these components" (36:503). The components are part of the Network Architecture Services (NAS) which "provides the objects and operations needed to construct robust real-time networks which support flexible, open architectures" (36:501). The use of SAS and NAS resulted in a "top-level software architecture definable in terms of standard system building blocks with well-defined behavior and interfaces and eliminates a major source of errors which come in executive logic control" (36:502). "The ability to rapidly construct a working system and focus on real application interfaces rather than system software inconsistencies coupled with NAS extensive support software and instrumentation resulted in an extremely successful effort" (36:514).

2.5 Conclusion

Shaw recognized that successful software designs can be grouped into broad, general categories, each representing a distinct software architecture. Several researchers, including Shaw, are convinced that the use of domain-specific architectures will lead to more reuse at the design level which should substantially increase reliability and reduce development costs for new software systems. The experiences of Softech with the U.S. Army, the SEI with their flight simulator model and the FODA methodology, Batory and O'Malley's hierarchical systems, the FDD at Goddard Space Flight Center with the GOOD methodology and TRW with CCPDS-R suggest that software architectures *can* facilitate development of large, complex systems.

III. Requirements Analysis[1]

3.1 Introduction

The wide availability of powerful, relatively low-cost computer hardware has led to an explosion in the demand for computer software products to automate a multitude of new tasks. Using traditional methods, computer scientists and programming professionals have been unable to meet, in a timely manner, this demand for the sophisticated, large-scale, reliable software systems required for these new applications. Clearly, a new approach to software design and construction is needed.

> Software engineering will evolve into a radically changed discipline. Software will become adaptive and self-configuring, enabling end users to specify, modify and maintain their own software within restricted contexts. Software engineers will deliver knowledge-based application generators rather than unmodifiable application programs. These generators will enable an end user to interactively specify requirements in domain-oriented terms.... and then automatically generate efficient code that implements these requirements. In essence, software engineers will deliver the knowledge for generating software rather than the software itself.
>
> Although end users will communicate with these software generators in domain-oriented terms, the foundation for the technology will be formal representations... Formal languages will become the lingua franca, enabling knowledge-based components to be composed into larger systems. Formal specifications will be the interface between interactive problem acquisition components and automatic program synthesis components.
>
> Software development will evolve from an art to a true engineering discipline. Software systems will no longer be developed by handcrafting large bodies of code. Rather, as in other engineering disciplines, components will be combined and specialized through a chain of value-added enhancements. The final specializations will be done by the end user. KBSE (Knowledge Based Software Engineering) will not replace the human software engineer; rather, it will provide the means for leveraging human expertise and knowledge through automated reuse. New subdisciplines, such as domain analysis and design analysis, will emerge to formalize knowledge for use in KBSE components. (26:629-630)

[1]This chapter was co-written with Captain Mary Anne Randour. It is included in AFIT Technical Report AFIT/EN/TR-92-5 and also appears in (33).

Perhaps this vision can become a reality for selected domains, not just within the next century as Michael Lowry predicts, but within the next few years. Research is currently underway at the Air Force Institute of Technology (AFIT) to achieve such a reality. Developing a full-scale application generation system, which is capable of automatically producing efficient code to satisfy user-specified requirements presented in domain-oriented terms, is a considerable task which will require several man-years of effort. However, one element of application generation, the combining or composing of required components into the proper framework or architecture, is attainable in the near term. This chapter explores the issues involved in developing such an end-user application composer and describes one possible methodology for accomplishing it.

3.2 Operational Concept

Several roles are discussed in describing this new approach to software development, an approach where the end-user generates a software application to satisfy his requirements using the software professional's knowledge about how to generate such applications. Some of these roles are new, others are relatively unchanged from those in traditional software system development.

1. System Analyst – Specifies new systems in a domain (20:4). Responsible for developing the concept of operations (defining policy, strategy, and use of application) and defining training requirements (10).
2. System Engineer – Works with the system analyst to partition the system into subsystems and assigns the tasks to software or hardware development, as appropriate (2).
3. Domain Engineer – Possesses detailed knowledge about the domain and gathers all the information pertinent to solving problems in that domain (20:4). Models the real-world entities required to satisfy the policy, strategy, and use of an application as defined by the system analyst. Determines how, if possible, these entities can be modeled within the constraints specified by the software engineer (10).
4. Software Engineer – Designs new software systems in the domain (20:4). Responsible for defining a formalized structure for the domain knowledge and providing the translation from the domain-specific terms to executable software (10).
5. Application Specialist – Uses systems in the domain (20:4). Familiar with the overall domain and understands what the new application must do to meet the requirements

(a sophisticated "user"). Provides the application-specific information needed to specify an application.

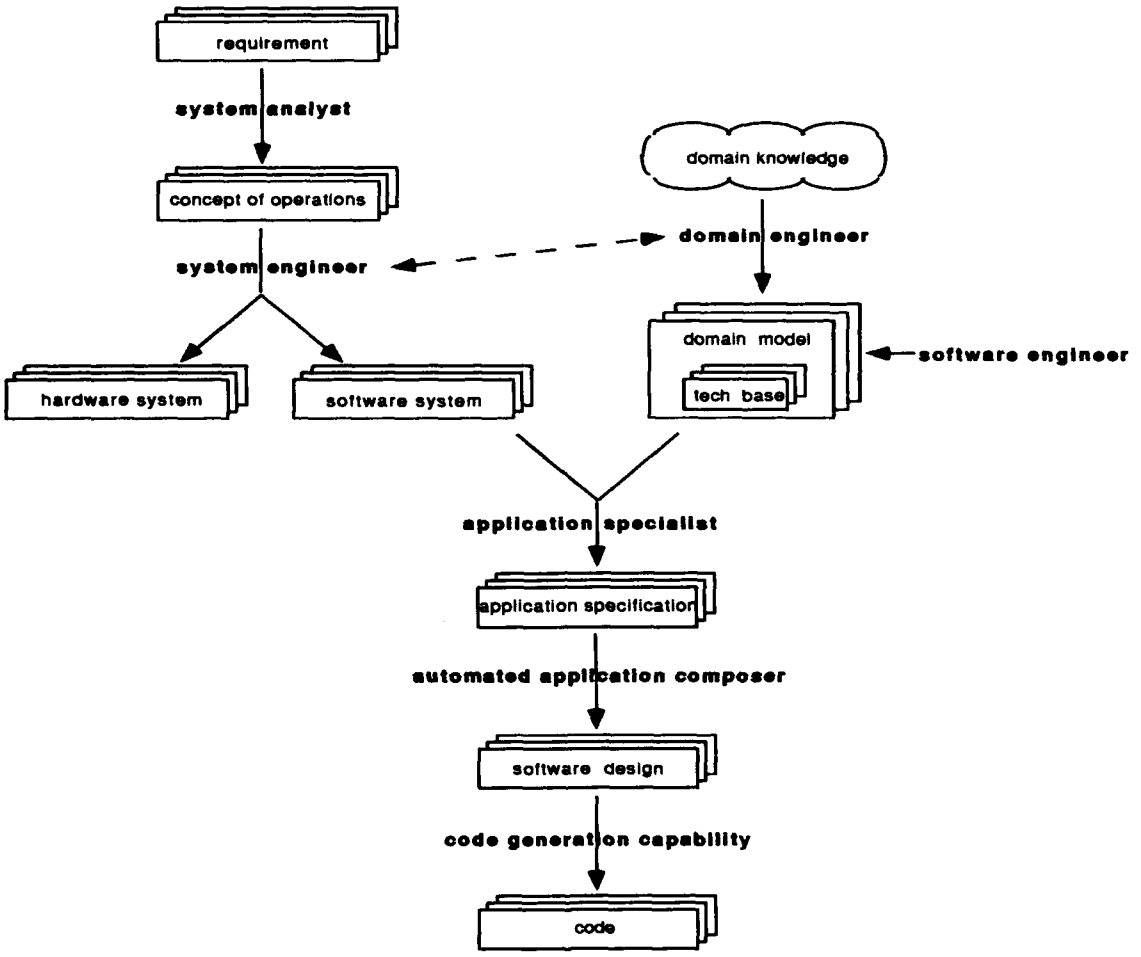

Figure 3.1. Roles

The relationships among these roles are shown in Figure 3.1. Usually, a new system begins with the identification of a new requirement. This requirement, if valid, is forwarded to a system analyst who develops a concept of operations. The system analyst works closely with the system engineer who partitions the system into software and hardware subsystems. The system engineer consults the appropriate domain engineer to define which components of his domain will be needed for software applications in the domain. The domain engineer and the software engineer decide on which components are needed to model the domain. The software engineer formalizes the domain knowledge provided

by the domain engineer into a domain model and its technology base. The application specialist, using the domain model established by the software and domain engineers, creates a specification for an application. From this specification, an automated application composer generates a software design which is then input to a code generation capability.

3.3 General System Concept

3.3.1 Overview An overview of the application composition system's components and their relationships to each other appears in Figure 3.2. First, domain analysis is performed, which consists of gathering appropriate domain knowledge, formalizing it via a domain modeling language, and storing it in a domain model. The structure of the domain model is determined, in part, by the domain modeling language (DML) chosen. The software architecture model, like the DML, imposes a specific structure on the domain model, on the grammar used by the application specialist, and, ultimately, on the final application specification. The domain model is used to develop a domain-specific grammar. Although it may be transparent to the application specialist, he actually uses two grammars: one to identify domain-specific information and one to specify the architecture of the application. The architecture grammar remains the same for different domains; only the domain-specific grammar changes. Application-specific data is written using these two grammars and is converted into objects in the structured object base by the parser.

The populated structured object base and information from the technology base are combined to build an executable prototype. First, the application specialist performs semantic checking on the structured object base to ensure all constraints on the system have been met. He then executes the prototype to demonstrate the behavior of the proposed application. If the prototype does not behave as required, the application specialist can change the original input and re-parse it into the structured object base. Using the knowledge encoded in the domain model and the software architecture model, the structured object base is manipulated into a formal specification for a domain-specific software architecture (DSSA). The DSSA *is* the system design and becomes the basis from which code is generated. A visual system provides a graphical representation of the structured object base and the DSSA, as well as a means to add to or modify them.

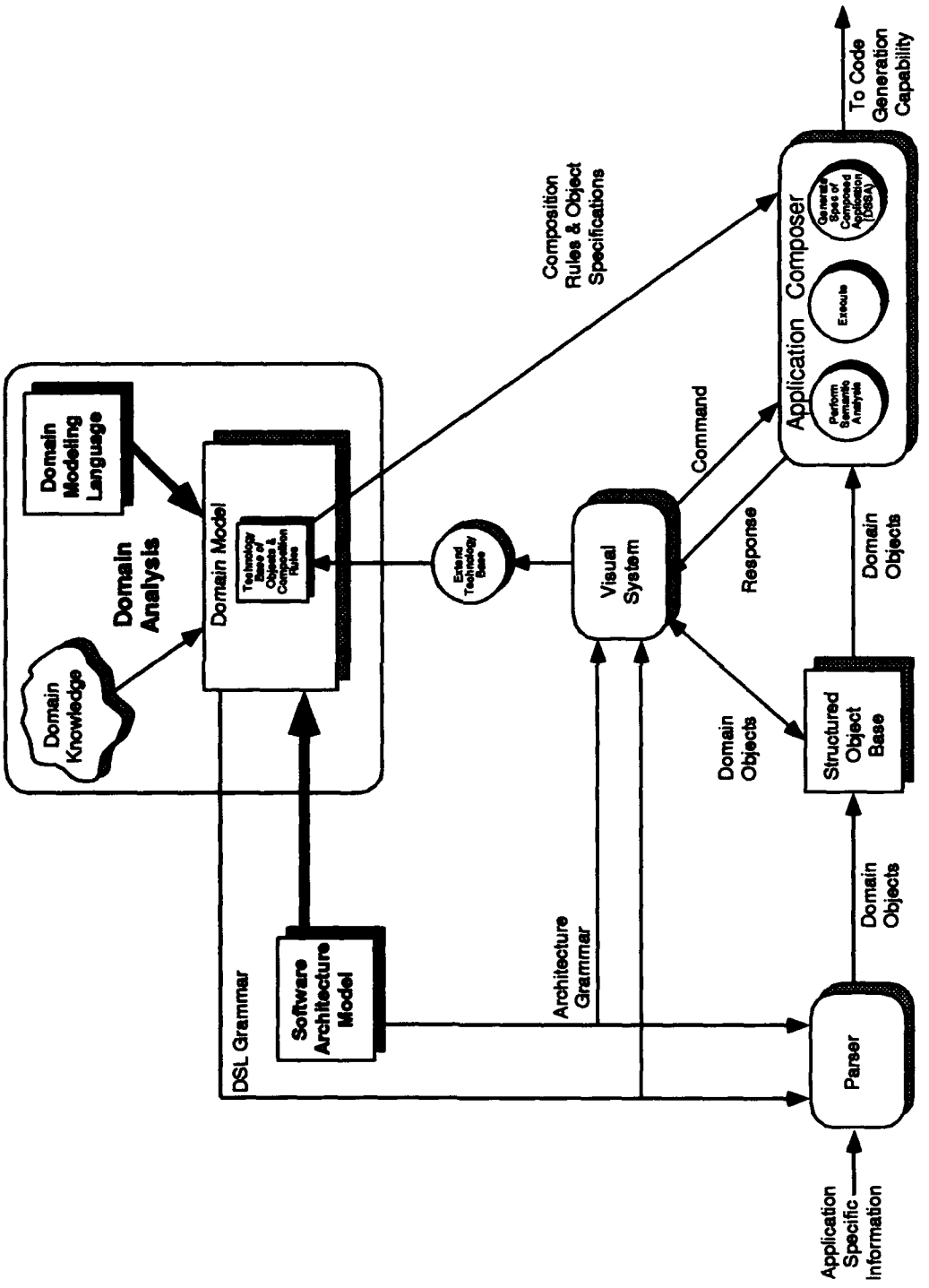

Figure 3.2. General System Overview

The remainder of this section describes the above concepts and activities in more detail.

3.3.2 Developing a Formalized Domain Model Before any applications can be composed using this proposed system, the domain must be analyzed and modeled. In the software engineering context, a domain is commonly defined as "an application area, a field for which software systems are developed" (31:50) or "a set of current and future applications which share a set of common capabilities and data" (20:2). Identifying the boundaries of the domain, as well as "identifying, collecting, organizing, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain" (20:2-3), constitutes domain analysis. Domain analysis is currently the subject of several other research efforts and is not directly addressed in this project. However, it is important to gather the basic data, formalize it, and store it in a standard format.

3.3.2.1 Domain Knowledge Domain knowledge is the "relevant knowledge" that results from a thorough domain analysis and later evolves naturally as more experience is gained solving problems in the domain (31:47). More specifically, domain knowledge consists of: basic facts and relationships, problem-solving heuristics, domain-specific data types, and descriptions of processes to apply the knowledge (4). In the context of this project, domain knowledge includes: descriptions of domain-specific objects (including their attributes and operations), data types, composition rules, and templates for commonly used architectural fragments.

3.3.2.2 Domain Modeling Language An analogy to a domain modeling language (DML) can be found in the more familiar data definition language of a database management system. A data definition language describes the logical structure and access methods of a database (21), just as our DML describes the logical structure of a domain model and defines how the objects can be accessed. A DML used to encode domain knowledge into a domain model must be able to formally describe:

1. Object Classes: Abstractions of real-world entities of interest in the domain.
2. Operations: Behavior of the objects in the domain.
3. Object Relationships and Constraints: Rules for relating objects (and sets of objects) to other objects, as well as the constraints on these relationships. Examples include:
   (a) Communication Structure: Message passing between/among domain classes and operations.
   (b) Composition Structure: Rules for combining domain object classes into higher-level application classes and operations into higher-level application operations.
4. Exception Handling: What to do when an error is encountered.

To be useful in an automated system, the domain knowledge must be encoded into a format that the software system can manipulate. This problem is analogous to encoding knowledge in an expert system, where human knowledge is gathered and represented as rules that allow a computer program to utilize the information. Neil Iscoe describes a method for encoding domain knowledge into a domain model (see (19) for details). He proposes using a domain modeling language or a meta-model as the basic framework to instantiate a domain model based on some operational goal(s) (reasons for which the knowledge will be used) (see Figure 3.3). Our operational goal is to "use the domain model, software architecture model, and structured object base to generate a software architecture for the application problem to be solved – to generate a domain-specific software architecture" (2).

3.3.2.3 Domain Model A domain model is a "specific representation of appropriate aspects of an application domain" (18:302) including functions, objects, data, and relationships (30). It is a result of expressing appropriate domain knowledge (identified by the domain engineer) in a domain modeling language with respect to certain operational goals (18:301-2).

Several researchers (5, 11, 12, 24) have indicated that software engineering must become more of an engineering discipline if we are ever to reap the benefits of design reuse (increased productivity, improved reliability, certifiability, etc.). When designing specific applications, engineers use models, "codified bodies of scientific knowledge and technology presented in (re)usable forms" (11:256) which are available to all practioners in various

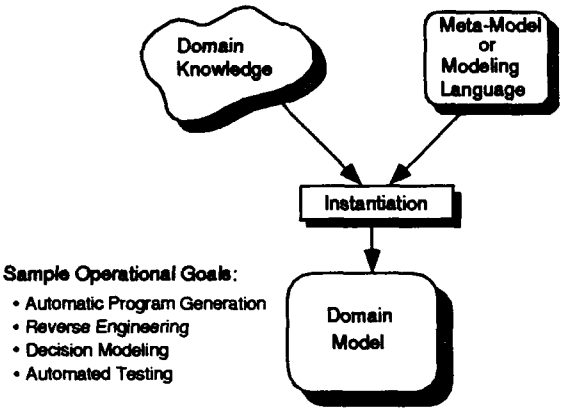

Figure 3.3. Domain Model Instantiation

technology bases. Reuse of these validated, commonly-used models, which are readily available in various technology bases, allows the engineer to construct a practical, reliable solution to the problem at hand.

Contained within our domain model is such a technology base which acts as a repository for our reusable models. In our system, these models are often referred to as components. Using an object-based perspective, a component can represent a real-world entity, concept or abstraction and encompasses all descriptive and state information for that entity/concept/abstraction as well as its behavior (what operations or functions it performs and/or what transformations it undergoes). Components can be primitive domain objects as described above or a "packaging" of these objects whose structure is determined by the software architecture model. These packaged components will be referred to as architectural fragments since they can be used to build an application architecture. The technology base contains templates for generic components, rules for component composition, and descriptions of primitive object behavior. The parameters required to instantiate these generic templates will be specified by the application specialist.

Domain analysis reveals common features of the software architectures that can be used to implement various specific applications within the domain. In addition, common constraints are identified and codified into rules used to determine how software com-

ponents can be legally combined. Using rules allows additional flexibility; any specific architecture can be built as long as it meets the criteria specified by the rules.

3.3.3 Building A Structured Object Base Several steps must be taken to build the structured object base. The following system components are essential to this phase.

3.3.3.1 Domain-Specific Language As with our domain modeling language, an analogy to a domain-specific language (DSL) can be found in a data manipulation language from the realm of database management systems. In the database context, a data manipulation language allows the user of a database to retrieve, insert, delete, and modify data stored in the database (21:13). In our context, a DSL is a language with syntax and semantics which represents all valid objects and operations in a particular domain, allowing modeling and specification of systems within that domain (32). According to James Neighbors, a domain language is a machine-processable language derived from a domain model. It is used to define components and to describe programs in each different problem area (i.e., domain). The objects and operations represent analysis information about a problem domain (28). In our research, a domain-specific language is defined as a formal language used to define instances of objects and operations specific to a domain.

The objective of our DSL is to generate the structured object base needed to specify an application architecture within a specific domain. To do so, it must be able to:

1. Instantiate objects
2. Instantiate generic objects
3. Instantiate generic architectural fragments
4. Compose the instantiated objects and architectural fragments in some meaningful way

The object classes defined in the domain model are merely templates or patterns to be used when constructing objects; they do not refer to specific, individual objects. The first sentence type listed above creates specific instances of the objects in the object base. These objects are used in building architectural fragments or as parameters for generics.

Default values can be used for attributes so these values need not be entered through the DSL every time they are used.

Generics, stored in the technology base, provide templates for commonly used objects and components; thus, the application specialist need not start from scratch each time he wants to include one of these commonly used components. Generics must be instantiated before they can be used. Instantiation is done by specifying which model is to be used and providing specific instances and/or other data, as required. For example, a generic architectural fragment may use three objects of a certain class. When this generic is instantiated, three specific object instances of the required class must be given.

3.3.3.2 Software Architecture Model In addition to identifying the objects to be used in generating a particular application, the application specialist must indicate what is to be done with those objects; i.e., he must identify the application operations. Domain primitive operations, associated with primitive objects, are available in the technology base. But how can these primitive operations be assembled (composed) into application-specific operations? What are the rules for composing these primitive operations into application operations? How can these rules be represented and implemented?

Software architectures provide insight into software system composition. In its most fundamental sense, an architecture is a recognizable style or method of design and construction. A software architecture has been defined as "a template for solving problems within an application domain" (40:2-2) or "the high level packaging structure of functions and data, their interfaces and controls, to support the implementation of applications in a domain" (20:3). It provides a mechanism for separating "the design of (domain) models from the design of the software" (10). This separation of domain knowledge from software engineering knowledge allows each type of engineer to concentrate on the issues relevant to his own area of experience, without becoming an expert in the other discipline. By focusing only on the design of the software, the software engineer is able to develop simplified packaging and control structures which can be reused across a wide variety of domains.

Because a software architecture serves as a structural framework for software development, we can expect it to provide a consistent representation of system components as well

as the interfaces between those components. A standard representation ensures that each component is developed in the same manner, eliminating many implementation choices and simplifying the development process. This standardization also results in consistent interfaces between all components, enabling them to be easily combined. This consistency of component representation and interfaces should provide a suitable and flexible framework for composing primitive operations into application-specific ones.

3.3.3.3 Architecture Grammar Certain portions of the application specialist's input are not dependent on any particular domain; rather, they depend on the software architecture model. These architectural aspects of the application can be specified using a grammar common to all domains, an architecture grammar. This grammar enforces the structure imposed by the software architecture model by defining valid sentences for packaging the primitive domain objects into architectural fragments to define an application architecture. These sentences will compose application operations using domain-specific components described by the domain-specific grammar and other application operations.

3.3.3.4 Parser After the application specialist specifies the application components using the domain-specific language and architecture language, the input must be parsed into objects in the structured object base. The parser generates specific object instances whose initial states are determined by the application specialist's input.

3.3.3.5 Structured Object Base The structured object base contains application specific information: specific instances of domain object classes with all appropriate attribute values for determining the object's state, as well as relationships for both domain objects and operations. The kinds of objects that might populate the object base and the overall structural framework of those objects (the shape of the abstract syntax trees) are established by the domain and software architecture models. The specific object instances and the actual structure of the object base are determined by the application-specific information provided by the application specialist using the DSL and architecture grammars.

3.3.4 Composing Applications The application composer generates the application architecture specified by the application specialist. This is accomplished by combining the appropriate instantiated domain objects from the structured object base in accordance with the domain composition rules. After the architecture is generated, its behavior can be simulated to demonstrate its suitability and correctness. It should be noted that the operations associated with each object in the technology base are certifiably correct; that is, individual objects are guaranteed to behave as required. However, the specific objects which are composed into the application may have been combined in such a way that the composed application may not behave as expected or required. When the application specialist is satisfied that the composed architecture is actually the one desired, he can generate a formal specification for the architecture which can later be used to develop a fully coded system.

3.3.4.1 Semantic Analysis After an application is identified, the next step is to ensure that the specified composition is appropriate; i.e., that it makes sense and meets the constraints imposed by the composition rules. This step is accomplished via a semantic analysis phase. As in programming language compilers, one aspect of semantic analysis is to verify that a syntactically correct construct, which satisfies the restrictions of the grammar in which it was written, is "legal and meaningful" (13:10). To be legal and meaningful, the proposed application must meet certain other composition restrictions: e.g., components must already exist before they can be used, an input to one component must be produced as an output from another component, etc. Another aspect of semantic analysis is to use knowledge about domain objects and typical system constructions to assist the application specialist in choosing the components needed and in combining them appropriately to create applications which behave as desired. Errors identified during the semantic analysis phase must be corrected before the composition process can proceed.

3.3.4.2 Execute A composed application architecture that passes all semantic analysis checks is legal and meaningful, but does it do what the application specialist wants it to do? The execute component of the application composer simulates the behavior of the architecture, using object operations which specify each component's behavior. This

behavior simulation may not be efficient or robust enough to serve as a full-scale operational system, but it provides the application specialist timely feedback on the correctness of the specified architecture. If the application is incorrect (i.e., it does not behave as required/expected), the application specialist reassesses the components which were used in the application and how they were combined, creating a new or editted application to satisfy his requirements. This ability to simulate execution behavior in this rapid-prototype manner assures the application specialist that the proposed application actually behaves correctly before a formal specification and fully-coded system are generated.

3.3.4.3 Generate Specification A legal, meaningful, and correctly composed application provides a software architecture which satisfies the application specialist's requirements for a particular application. The software architecture can be used as a blueprint, template, or specification from which to design and implement a full-scale, operational version of the application. The generated specification is intended to be in a formal, machine-processable format which can be used directly by a code generation tool to produce a fully-coded application. However, the specification format could be tailored to provide whatever form is appropriate for the using organization: graphical, textual, etc.

3.3.5 Extend Technology Base Eventually, the technology base, which formalizes the knowledge about domain objects, will become outdated as understanding of the domain evolves and as the domain itself adapts to accommodate a changing technological environment. Although the technology base may appear to be static, it must be dynamic enough to accommodate this additional information as well as higher-level object classes and operations, generic components and architectural fragments that are developed. These additional elements give added flexibility to the application specialist because more predefined components are available for future applications

A specialized set of tools allows the technology base to be modified or extended to include this additional or revised domain knowledge. The extender must enforce the structure dictated by the domain modeling language and the software architecture model.

3.3.6 Visualization "A picture is worth a thousand words." This old adage is still true today, especially when dealing with complex and abstract concepts. The visual system provides the application specialist with a graphical view of the structured object base, as well as the application software architecture generated to satisfy his requirements. By reviewing these "pictures," the application specialist can more fully understand the components available for composition and the application just composed. Moreover, the visual system will also be capable of inserting new instances of domain objects into the structured object base, editing domain objects already in the object base, and executing the application composer. It also provides the capability to extend the technology base, enabling the application specialist and/or the software engineer to add/modify domain object classes, add/modify generic components, and add/modify architectural fragments.

The visual system is addressed in more detail in (44).

3.4 Related Research

Several other research efforts have addressed various aspects of the system we are attempting to develop. This section summarizes this related work and analyzes the similarities to and differences from our project.

3.4.1 Hierarchical Software Systems With Reusable Components Don Batory and Sean O'Malley are working to incorporate an engineering culture into software engineering. The traditional engineering mindset dictates that new systems are created by fitting well-tested, well-defined, and readily available building blocks into a well-understood blueprint or architecture, which, if properly used, is guaranteed to produce the desired system. To this end, they have developed a "domain-independent model of hierarchical software design and construction that is based on interchangeable software components and large-scale reuse" (5:2).

In Batory and O'Malley's view, each interchangeable component consists of an interface (everything externally visible) and an implementation (everything else). Different components with the same interface belong to a realm. All the components in a realm are considered to be interchangeable or "plug-compatible" (5:3) because they have identical

interfaces. Symmetric components have at least one parameter from their own realm and can be combined in "virtually arbitrary ways" (5:2) (also see Figure 3.4). Conceptually, components are seen as layers or building blocks for an application; a system is seen as a stacking of components, i.e., a composition of components. Constraints on stacking components (i.e., rules of composition) are derived from the compatibility of their interfaces.

Hierarchical software system design recognizes that constructing large software systems is a matter of addressing only two issues: which components should be used in a construction and how those components are to be combined together (5:16). It employs an open software architecture, which is limited only by the inherent ability of the components to be combined, i.e., by their interfaces. Symmetric components have no inherent composition restrictions; thus, composition rules are simplified while ensuring maximum design flexibility and potential reusability of components.

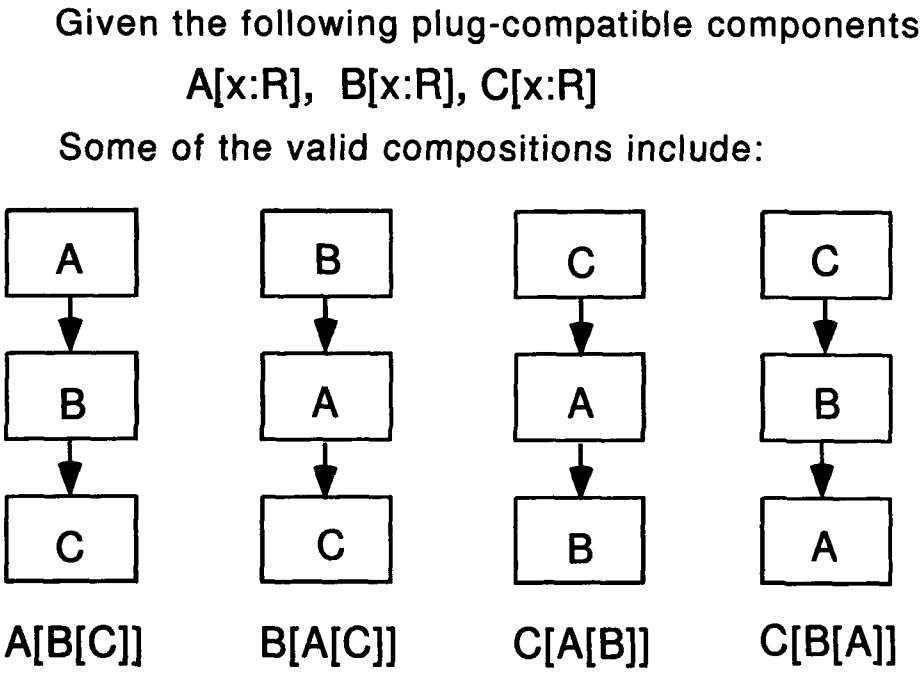

Figure 3.4. Combining Plug-Compatible Components

Concept	Grammar
Parameterized Components	Productions with non-terminals on right
Parameterless Components	Productions that only reference terminals
Symmetric Components	Recursive production
Component Interface	Left side of a production
Implementation	Right side of a production
Realm	Set of all productions with the same head
Software System	Sentence
Rules of Composition	Semantic error checking

Table 3.1. Analogy to Grammar

Batory and O'Malley use an interesting analogy, equating their concepts to a grammar, as shown in Table 3.1 (5:5). Using this analogy, a domain is a language. Consider the following example (5:5):

S = {a, b, c}	$S \rightarrow a \mid b \mid c$
R = { g[x:S], h[x:S], i[y:R]}	$R \rightarrow gS \mid hS \mid iR$

A realm S, having a set of components (a, b, and c), corresponds to a production where the non-terminal S can be replaced by either a, b, or c. Whenever a component from realm S is needed, a, b, or c could be used, depending on the behavior and level of detail needed. A realm R, whose components g, h, and i require parameters from realms S, S, and R, respectively, can be represented by a production where a non-terminal can be replaced by both a terminal and a non-terminal. The non-terminals on the right-hand side are the realms from which the parameters are provided. The complete analogy is summarized in Table 3.1.

Batory and O'Malley's work provides support for our research. It confirms the underlying principle of an application generator: building software systems from reusable components is "simply" a matter of selecting which components to use and deciding how

to compose them together. It reinforces our intention to use an object-oriented approach in designing our system. It also illustrates the role of component interfaces in system composition and demonstrates the importance of consistent interfaces and composition styles in developing rules for combining components.

On the other hand, the Batory/O'Malley work falls short, in some ways, of what we are attempting. It does not incorporate a mechanism for an application specialist to specify new applications in domain-specific terms; this is a primary emphasis of our project. It also does not seem to provide for tailoring of component composition to suit the application being built; composing component A with component B into component C will always produce the same behavior for C. We want to be more flexible in our compositions and allow A and B to be composed into C in one situation and C' in a different situation, depending on how the application specialist specifies the composition.

3.4.2 Automatic Programming Technologies for Avionics Software The Lockheed Software Technology Center has developed the Automatic Programming Technologies for Avionics Software (APTAS) system pictured in Figure 3.5 (25:2). The APTAS system, built for the target tracking domain, "takes a tracking system specification input via user interface with dynamic forms and a graphical editor, and synthesizes an executable tracker design" (25:1). An application specialist defines a new tracking application by answering questions which appear in pop-up, menu-like forms. His answers determine which additional questions are to be asked as he is guided through specifying a new tracker. When all pertinent specifications have been entered (defaults exist for questions which are left unanswered), the application specialist generates a software architecture for the new tracker via the architecture generator. A graphical user interface provides a "picture" of the application architecture and allows the user to change it interactively. After the application specialist is satisfied with the architecture just created, he generates executable code to implement that architecture via the synthesis engine (25). He can also invoke a run-time display which facilitates testing and analyzing the tracker just created.

The Tracking Taxonomy and Coding Design Knowledge Base is at the center of the APTAS system. It contains the system's specification forms, the primitive modules from

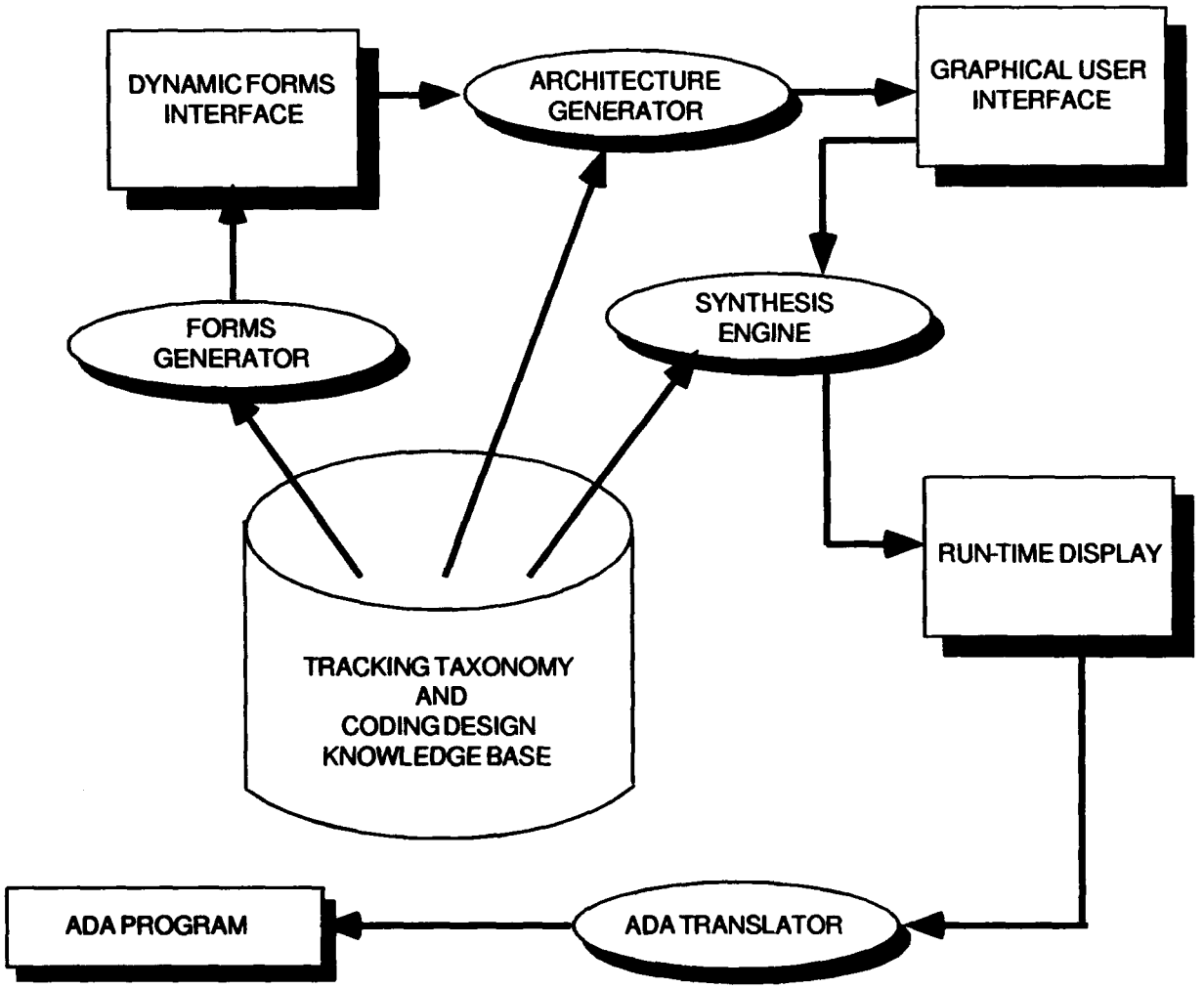

Figure 3.5. APTAS

which new trackers are constructed, and the composition rules which establish how primitive modules are to be combined. The application specialist's answers to the questions on the specification forms progressively reduce the number of primitive modules which are candidates for incorporation into the new tracker. The architecture generated upon completion of the forms specification is synthesized into an executable intermediate language, Common Intermediate Design Language (CIDL). The CIDL code can be executed to demonstrate system behavior. If the system behaves as desired, the CIDL representation can then be transformed into Ada code. The use of an intermediate representation, such as CIDL, localizes the code translation function and enables languages other than Ada to be targeted more easily.

The APTAS primitive modules and their composition rules are also written in CIDL. Extending the system involves writing new primitive modules and incorporating references to these new modules into the appropriate composition rules and specification forms. This is generally considered to be a software engineer's task (rather than an application specialist's), as CIDL is a software specification language and few tools exist to simplify the process.

APTAS is strikingly similar to the system we envision. It clearly demonstrates that the concept of user-initiated composition and generation of domain-specific systems is feasible. It allows application specialists to specify new applications in domain-specific terms, by way of menu-like specification forms. It also provides a sophisticated graphical user interface which can be used to construct and/or edit the tracker system, as well as to view the structure of the architecture.

There are, however, some major differences between APTAS and the system we are developing. APTAS's use of a domain-specific language is implicit and embodied in its graphical user interface. Our domain-specific language, on the other hand, is explicit and its grammar is usable in both textual and graphical modes. We believe this provides advantages to both the software engineer and application specialist in terms of adaptability, flexibility, and ease of use. In addition, APTAS currently lacks a set of convenient tools to facilitate extending its knowledge base; such a toolset is an integral part of our system.

3.4.3 Model-Based Software Development The Software Engineering Institute's (SEI) Software Architectures Engineering (SAE) Project has proposed a concept called Model-Based Software Development (MBSD) (24). Like Batory and O'Malley, MBSD strives to apply traditional engineering principles to software development by exploiting prior experience to solve similar problems. This prior experience is codified in models, "scalable units of reusable engineering experience" (24:11), which are stored in a technology base. In a mature engineering domain, the technology base will contain "all the components an engineer needs to predictably solve a class of problems, and the tools and methods needed to predictably fabricate a product from the components specified by the engineer" (24:4). Under MBSD, software development follows the engineering paradigm: reuse existing, mature models rather than starting from scratch for each new development. This involves much more than code reuse; the requirements analysis, design, and software architecture are reused each time the corresponding model is used.

MBSD uses a technology base, a repository of models and composition rules that share common engineering goals. Each model is mapped to a specification form and a software template for the target application language. The specification form is a text-based description which uniquely identifies a specific instance of a model. The software template is code containing place holders, which are replaced with information from the specification form (24:10).

As part of MBSD, the SEI uses the Object-Connection-Update (OCU) model as a consistent pattern of design, a software architecture. This model is especially suited to domains where the real world can be modeled as a collection of related systems and subsystems (24:17). Partitioning a system into subsystems provides different levels of abstraction, giving the flexibility to replace a subsystem with another that either provides a different function or has a different level of detail. In the OCU model, subsystems consist of a controller, a set of objects, an import area, and an export area as pictured in Figure 3.6 (24:18).

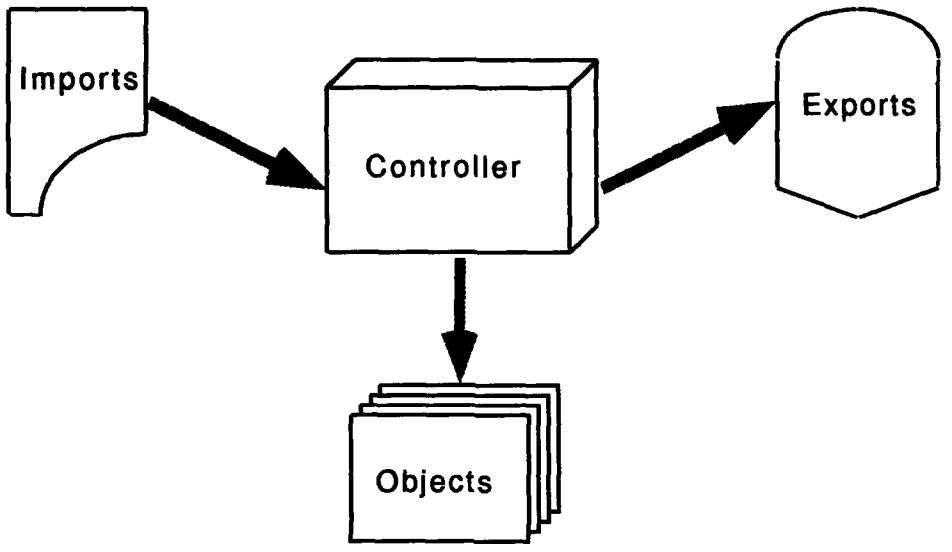

Figure 3.6. OCU Subsystem Construction

1. Controller – Performs the mission of the subsystem by requesting operations from the objects it connects. A controller is passive, triggered by a call to perform its mission, and depends on the other subsystem components to accomplish that mission.
2. Objects – Model behavior of real-world entities and maintain individual state information. An object is passive, triggered by a call from the controller to which it is connected.
3. Import Area – Makes data external to the subsystem available to the controller and its objects.
4. Export Area – Makes data internal to the subsystem available to the other subsystems.

Both controllers and objects have standard procedural interfaces used by external controllers or application executives to invoke some action. Controllers have the following procedures (24:19):

1. Update – Updates the OCU network based on state data in the import area and furnishes new state data to the export area.
2. Stabilize – Puts the system in a state consistent with the current scenario.
3. Initialize – Loads the configuration, creates objects, and defines the OCU network.

4. Configure – Establishes the physical connection between import area and input data as well as export area and the output data.
5. Destroy – Deallocates the subsystem.

All objects have procedures analogous to those for controllers, but operating on a single object instance. Specifically, these procedures are (24:20):

1. Update – Calculates the new state based on input data and the current state.
2. Create – Creates a new instance of the object.
3. SetFunction – Changes or redefines the function used to calculate the state.
4. SetState – Directly changes the object's state.
5. Destroy – Deallocates the object.

These well-defined and consistent interfaces for controllers and objects facilitate and simplify the application composition process.

MBSD provides some significant insights upon which to base our research effort. Its focus on the reuse of validated, engineering experience is attractive and we have adopted the notion of storing such information in a technology base. The OCU model provides a realistic approach toward composing primitive objects into application-specific subsystems.

3.4.4 Extensible Domain Models[2] The Kestrel Interactive Development System (KIDS) is a knowledge-based system that allows for the capture and development of domain knowledge (39). The representation of the domain knowledge constitutes a domain model, and these domain models are called domain theories. Essentially, the domain theory provides a formal language, natural to specialists in that domain, for specifying the problem they want to solve. The KIDS system provides support for constructing, extending, and composing domain theories, and over 90 theories have been built up in the system (39). Additionally, the set of domain theories developed during the domain modeling effort serves as the basis for software synthesis.

The foundations of the KIDS approach emerged from years of research into the specification and synthesis of programs (39). Concepts from algebra and mathematical

[2]This section was provided by Major Paul D. Bailor

logic are used to model application domains and synthesize verifiably correct software. Domain modeling entails the analysis of the domain into the basic types of objects, the operations on them, and their properties and relationships. The domain model is then expressed as a domain theory. Theories are useful for modeling application domains for the following reasons.

1. The basic concepts, objects, activities, properties, and relationships of the domain are captured by the types, operations, and axioms of a theory.
2. Any queries, responses, situation descriptions, hypothetical scenarios, etc. are expressed in the language defined by the domain theory.
3. The semantics of the application domain are captured by the axioms, inference rules, and specialized inference procedures associated with the domain theory.
4. Simulation, query answering, analysis, verification of properties, and synthesis of code are supported by inference within the domain theory.
5. Various operations on models such as abstraction, composition, and interconnection are supported by well-known theory operations of parameterization, importation, interpretation between theories, and others. Thus, a high degree of extensibility is obtained.

3.5 Specific System Concept

Several aspects of the system described in Section 3.3 depend heavily on the choice of the models and tools used in the implementation. These selections may impact other parts of the system. Figure 3.7 is a modification of the system overview, incorporating the specific models and tools to be used. It represents Architect, the specific system which is to be implemented during this research effort.

3.5.1 System Overview Figure 3.7 illustrates how specific tools and models further define Architect. REFINE, as the domain modeling language, imposes its structure on the domain model (which will be represented in REFINE also). Input, written in the domain-specific and architecture grammars, is processed through a parser generated by DIALECT. DIALECT requires two inputs to generate a parser: a DIALECT domain model (a subset of the system domain model) and a grammar definition. The DIALECT parser creates abstract syntax trees in the structured object base. The visualizer will be implemented

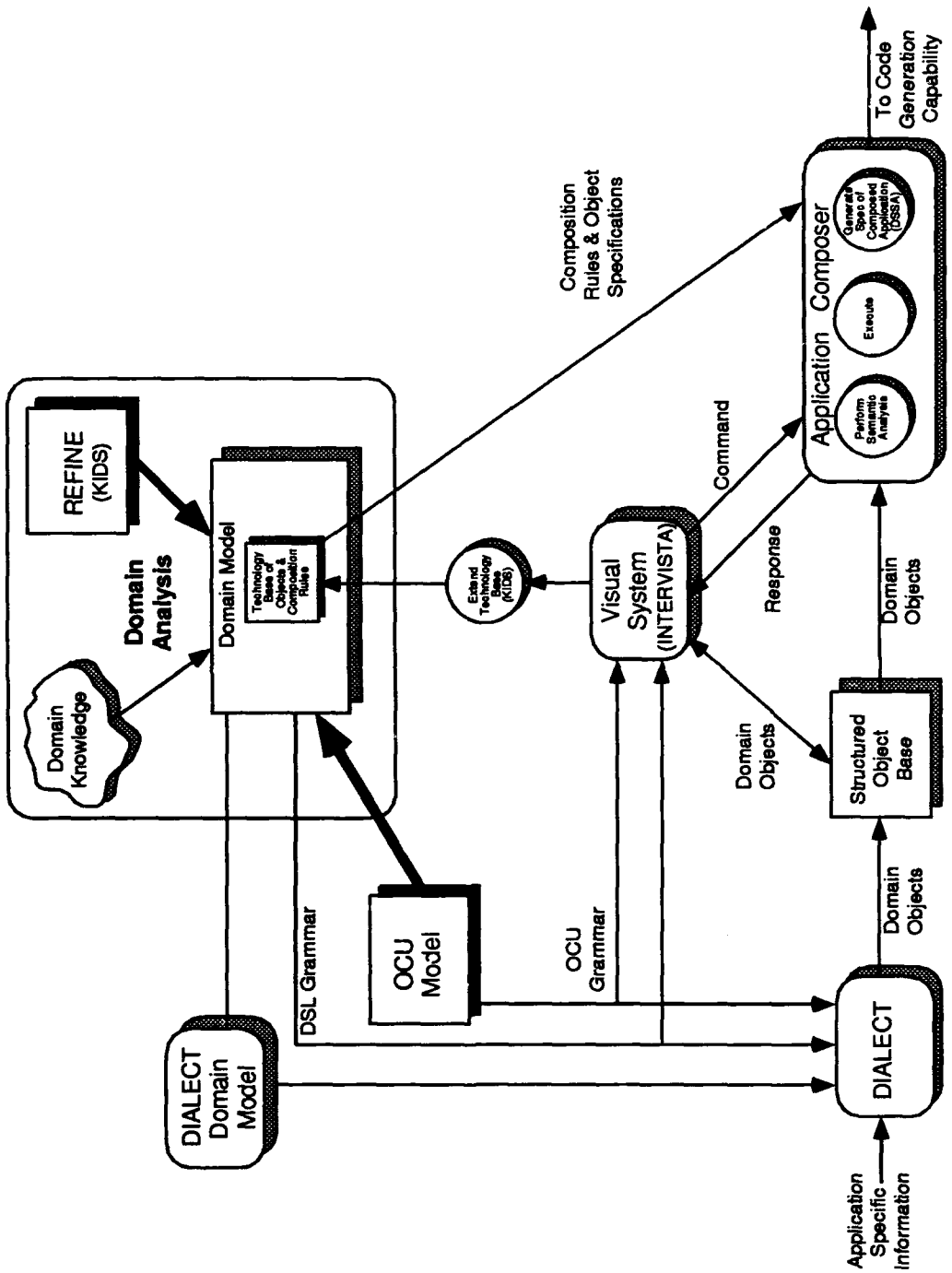

Figure 3.7. Overview of Specific System

using INTERVISTA. The SEI's OCU model will serve as our software architecture model, providing a structure around which to generate our applications. KIDS will serve as a mechanism for realizing extensibility of the domain model and technology base.

3.5.2 Software Refinery Software Refinery is a formal-based specification and programming environment developed by Kestrel Institute and available commercially from Reasoning Systems, Inc. We have selected this environment in which to implement Architect for several reasons, but the main factor in our decision is REFINE's powerful, integrated toolsets that allow rapid prototyping. This decision has many implications on how the system will operate, as we will show.

3.5.2.1 Capabilities The REFINE environment consists of the following tools:

1. A programming language (REFINE) which includes set theory, logic, transformation rules, pattern matching, and procedures (35:1-2). The REFINE language provides a wide range of constructs from very high level to low level, making it suitable for various programming styles, including use as an executable specification language.
2. An object base which can be queried and modified through REFINE programs (35:1-2). "Object classes, types, functions and grammars are among the objects you can define and manipulate" (35:1-4) with several built-in and powerful object base manipulation tools.
3. A language definition facility (DIALECT) which allows design of languages using an extended Backus Naur Form notation. REFINE supplies a lexical analyzer, parser, pattern matcher, pattern constructor, and prettyprinter for the language (35:1-2).
4. A toolset (INTERVISTA) which is useful in creating a visual, window-based interactive user interface.

3.5.2.2 Domain Modeling Language Some domain modeling languages already exist for expressing domain knowledge within a formalized domain model; we considered two such languages: the Requirements Modeling Language (RML) and REFINE.

RML was designed as a research tool as part of the Taxis Project at the University of Toronto. It allows "direct and natural modeling of the world" (16:3) in an object-oriented manner which "captures and formalizes information that is left informal or not documented in current approaches" (16:1). RML can express "assertions (what should be true in the

world), as well as entities (the 'things' in the world) and actions (happenings that cause change in the world)" (16:4). This is precisely the type of information we want to capture in our domain model.

Even though both RML and REFINE appear to be capable of expressing the kind of information we require in the domain model, we chose REFINE as our domain modeling language for the following reasons:

1. REFINE provides an integrated environment including programming constructs and powerful object base manipulation tools. Use of REFINE's existing tools eliminated the need to write our own, allowing more time to be spent on the research itself.
2. RML is not an executable language; no compilers currently exist. To use RML, we would be forced to develop a compiler, a considerable overhead to our project. As REFINE is also capable of expressing the information we require, it is unclear what added benefits RML could provide to justify this additional expense.
3. The REFINE environment includes compatible tools (DIALECT and INTERVISTA) useful in other portions of the system.
4. REFINE is a commercially available and supported product.
5. Members of the research team already possessed a working knowledge of REFINE.

3.5.2.3 Parser "DIALECT is a tool for manipulating formal languages" (34:1-1). A part of the REFINE software development environment, DIALECT generates appropriate lexical analyzers, parsers and pretty-printers for user-specified, context-free grammars. Valid input is parsed and stored as abstract syntax trees in the REFINE object base, according to the structure established in the DIALECT domain model. The DIALECT domain model defines object classes, object attributes, and the structure of the instances in the object base. DIALECT also supports grammar inheritance, allowing for a base language with several variations or "dialects." In Architect, the architecture grammar acts as the common base, and the domain-specific grammar specifies a particular variation. DIALECT does impose restrictions on the grammars. Since DIALECT generates an LALR(1) parser, the grammar must be consistent with this type of parser. Also, the productions in the grammar must correspond to the structure defined in the domain model. Altering some productions may require updating the DIALECT domain model.

3.5.2.4 Structured Object Base The structured object base was implemented using the REFINE object base. REFINE includes many tools which, when combined with REFINE code, provide all of the functions necessary to manipulate the structured object base. However, the object base must be accessed through the REFINE environment.

3.5.2.5 Technology Base Models in the technology base were represented as REFINE code and stored in REFINE's object base. Although separate conceptually, the technology base and structured object base are not physically separate. Access is controlled by Architect to avoid any confusion.

3.5.2.6 Visual System INTERVISTA provides a tool set with which to generate a window-based graphical user interface. It is compatible with the other REFINE tools; therefore, it is easily integrated. INTERVISTA can access the REFINE object base, so all its required data is readily available.

3.5.3 Object-Connection-Update Model We have selected the Software Engineering Institute's Object-Connection-Update (OCU) model for our software architecture model. As such, it provides a framework for composing applications – a standardized pattern of design for all applications and their components. The OCU model's consistent interfaces enable all components to be accessed in the same manner and its intercomponent communication scheme ensures that each component can readily access the external data needed for its processing. Currently, our focus is on implementing the subsystem aspect of the OCU model; the hardware interface portion of the model will be addressed in follow-on research efforts.

The choice of the OCU model for our software architecture model had certain implications for Architect.

1. Terminology – In keeping with the OCU model, we will refer to domain primitive objects as "objects," compositions of objects as "subsystems," the locus of control of a subsystem as a "controller," and the overall application itself as an "executive" (see Sections 4.2.1.3 and 4.2.2.1 for a more detailed discussion of the executive). External data needed by an object are "input-data," whereas data to be made externally available are "output-data." An "import area" serves as a focal point for all external

data needed by the subsystem and an "export area" is the focal point for all internal data to be made available to other subsystems. The OCU model's names for the object and controller procedural interfaces have also been retained.

2. Use of a Technology Base – Although the concept of storing reusable domain knowledge or models in a technology base is not unique to the OCU model, it is a fundamental component of Model-Based Software Development of which the OCU model is a part.

3. Domain Analysis – The OCU model deals with objects and subsystems. This imposes a constraint on the domain engineer and will impact the manner in which domain analysis is conducted. Under the OCU model, the domain engineer must model the domain in terms of subsystems which can be composed from lower-level, more primitive objects. Many domains can be naturally modeled in such a way; with other domains, a new mindset may be needed to incorporate the subsystem/object requirements of the OCU model. Alternatively, an additional class of software architectures may need to be defined.

4. Definition of Domain Objects – The OCU model requires that all objects be defined in the same manner. Each object has state data, other descriptive information, input-data/output-data definitions, and the following procedural interfaces: Update, Create, SetFunction, SetState, and Destroy. These requirements dictate how the objects will be constructed, severely limiting implementation choices. However, it is this very limitation which provides the flexibility that allows the domain objects to be successfully composed to satisfy the application specialist's specification.

5. Definition of Architectural Fragments – The OCU model requires that all architectural fragments (subsystems) be described in the same way. All subsystems have an import area, export area, controller, and objects. Each controller has the following procedural interfaces: Update, Stabilize, Initialize, Configure, and Destroy. As with the objects, this apparent limitation on implementation choices actually provides great flexibility in composing subsystems and combining them into a complete application.

6. Composition Rules – The standardized object/subsystem definitions and interfaces of the OCU model simplify application composition. There are no inherent restrictions preventing one component from being combined with another; all composition rules are domain-specific and do not derive from the software architecture.

7. Intercomponent Communication – The OCU model establishes and enforces a standard method for intercomponent communication. Communication external to the subsystem is localized in the import area which obtains the necessary input-data for all objects within the subsystem. This localization of communication concerns within the narrow guidelines imposed by this scheme simplifies intermodule communication: subsystems can readily obtain needed external information in a consistent manner and changes in the low-level implementation of the communication process are hidden from the subsystems/objects.

8. Structure of the Resulting Application Specification – Obviously, the specification produced by the application composer is impacted by the choice of a software ar-

chitecture model. The OCU model produces an application (an "executive") which is composed of subsystems. These subsystems can be decomposed into objects and lower-level subsystems, if appropriate. This hierarchical structure is preserved in the generated specification.

The OCU model is the result of years of research and experimentation by the SEI. It has been used successfully in the flight simulator, missile, and engineering simulator domains (10) and appears to provide a suitable structure for composing applications within our application composition system.

3.6 Conclusion

Software engineering may be on the brink of a new era, an era in which software engineers develop knowledge about generating software systems and application specialists actually create the software systems using familiar, domain-oriented terms. Our research, which builds on important work already accomplished by various researchers, is designed to demonstrate the feasibility of such an application composer.

IV. Software System Design Overview

This chapter presents an overview of the high-level design of the application composition system introduced in Chapter III (Architect). It discusses various preliminary design decisions (which stem from the choice of the OCU model for the software architecture of the composed applications) by reviewing the OCU model and identifying certain adaptations which were made for this implementation. In addition, the implementation's goals/objectives, conventions, and data structures are examined.

4.1 High-Level System Design

This section[1] describes the high-level design of Architect, the system presented in Section 3.5.

4.1.1 Design Goals Throughout the design process, an attempt was made to optimize several fundamental goals. These goals include:

4.1.1.1 Domain Independence Since Architect must be applicable to any domain, it should not directly incorporate (i.e., "hardcode") knowledge about a specific domain or type of domain; the technology base is the proper, sole repository for such domain-specific information. If any domain knowledge were to be included in Architect, code changes would likely be required before it could be used with a new or modified domain. Obviously, this greatly limits the applicability and usability of the system.

4.1.1.2 Extensibility It would be very naive to assume that an initial domain analysis will reveal all possible knowledge about a particular domain and that the domain model, which formalizes this knowledge, will never change. In reality, the domain model will continue to evolve as existing knowledge is further refined and/or new domain information is added to the system. If this evolution cannot be achieved easily, Architect will quickly become obsolete.

[1]This section was jointly written by Captains Cynthia Anderson and Mary Anne Randour. It is included in AFIT Technical Report AFIT/EN/TR-92-5 and also appears in (33)

4.1.1.3 Flexibility Because the concept of application composers is rather new, we do not yet know how application specialists and software engineers will best be able to use them. Architect, therefore, must be flexible enough to allow multiple methods for performing various tasks and a wide range of application specification options.

4.1.1.4 Usability Application specialists, the primary users of this system, must have some degree of software programming knowledge, but they can not be expected to have the same degree of understanding as a software engineer. Therefore, it is important that the system not require detailed programming knowledge from its users. In many cases, the goals of usability and flexibility conflicted, so a balance had to be found.

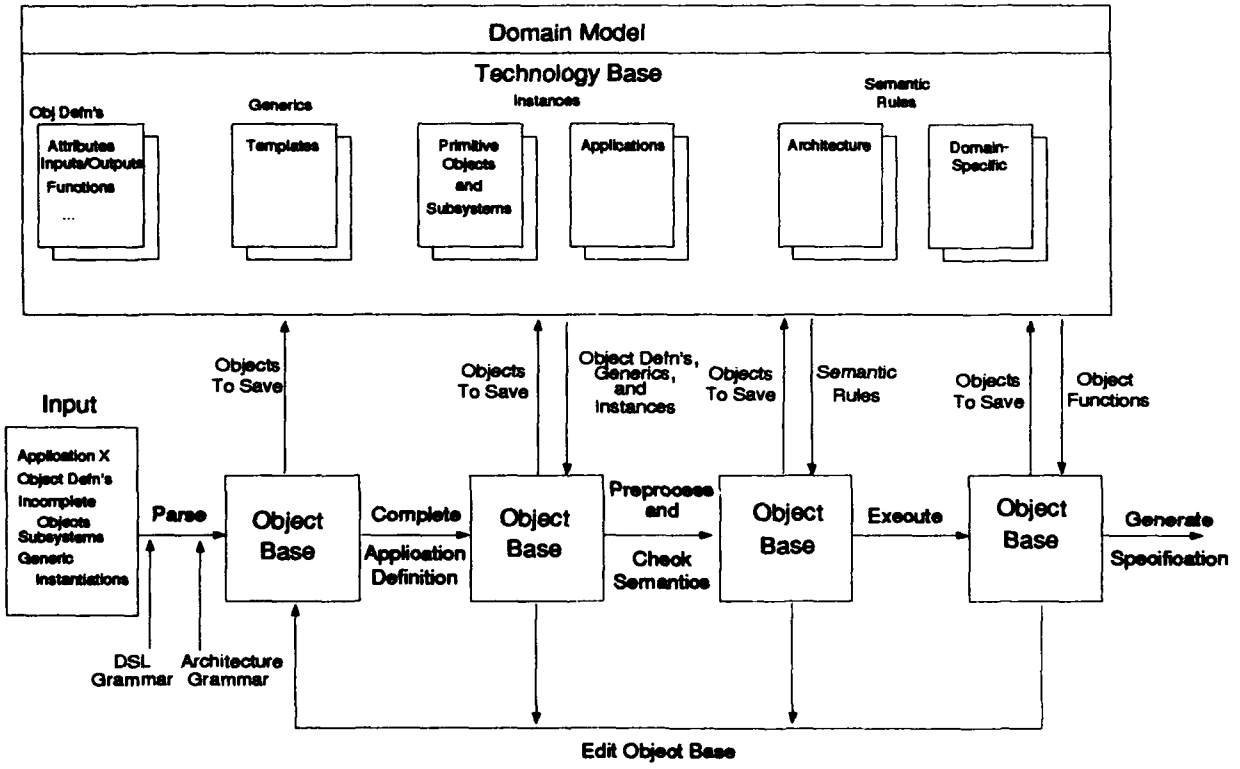

Figure 4.1. System Operations

4.1.2 Concept of Operations The steps which are followed when using this application generator are depicted in Figure 4.1 as labels on the flow arrows. The application specialist must first identify all objects to be used in the application specification and enter (parse) them into the structured object base using domain-specific and architecture

grammars. Some of these objects may require further information before they are fully defined (e.g., previously saved objects must be located and loaded, "holes" in generic object templates must be filled, etc.); the application specialist provides this additional information by completing the application definition. Although the application definition may be considered complete from the user's point of view, some data needed by the system may not yet be directly available; preprocessing the application specification automatically generates this essential information. When the application is fully defined, semantic checks are performed to identify any composition errors, which must be corrected before the application's behavior can be simulated. At any time, the application specification can be changed (edited), usually in response to a semantic error or to include additional data. If no semantic errors exist and no additions/changes have been made, the application's behavior can be simulated (executed). If the application behaves as the application specialist intended, he may generate a formal specification for the composed application which will be used by an automatic code generator to produce a fully realized application.

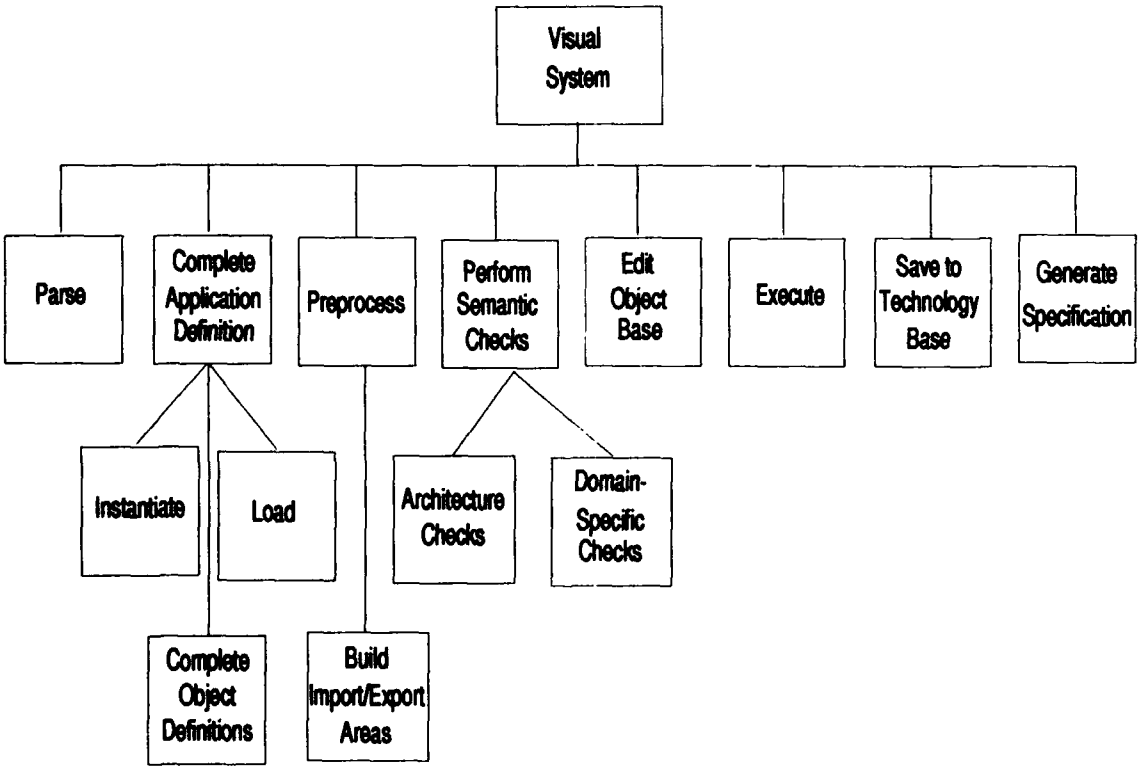

Figure 4.2. System Structure

4.1.3 Software System Design The eight steps outlined above correspond directly to Architect's eight top-level modules as shown in Figure 4.2. The highest level module, the visual system, will eventually control each of the other modules as well as all user interactions through a graphical interface. However, the basic system was developed before the visual system was completed; therefore, a simple user interface, which is easily replaced, was implemented. In the system design, we have made a conscious decision to keep application specification a domain-oriented, rather than programming-oriented, process. The system is designed to use all available domain knowledge to insulate the application specialist, as much as possible, from programming details, conventions, and jargon.

Each of Architect's major functions is encapsulated into one of the system's top-level modules. These modules are further discussed in the remainder of this section.

4.1.3.1 Parse Using REFINE, data can be input into the object base using one of two different methods: through a grammar or directly by using built-in REFINE functions.

Using the DIALECT tool allows the application specialist to reuse his input files as templates for other application definitions. The grammar also provides a consistent format for saving objects from the object base into the technology base. The domain-specific portions of the grammar can be separated from the architectural components. DIALECT allows grammars to inherit the productions of other grammars. In this case, each domain-specific grammar inherits the same architecture grammar. If the domain is changed, only one grammar is affected. However, the application specialist must conform to the structure imposed by the grammars. If the current domain changes, the domain-specific grammar will require appropriate, corresponding changes. If a different domain is to be used, a new domain-specific grammar must be written; however, grammars for other domains can serve as a guide to facilitate creating new grammars.

An alternative approach is to build REFINE tools that allow the application specialist to interactively enter the objects into the object base. This method migrates most easily to the visual interface planned for a follow-on project (44). Also, this method is domain-independent. However, additional code must be written to save portions of the object

base. The developer must devise a standard format for the files to allow this data to be read back into the object base.

The best approach is to combine the two methods. The application specialist can input objects into the object base either through a grammar or interactively. The grammar provides a format for saving and retrieving all objects in the object base and a means of saving "templates" for application definitions. The interactive portions extend more readily to the visual system.

4.1.3.2 Complete Application Definition After all of the application specialist's input is parsed into the object base, additional processing is needed to complete the definition. The application specialist can fully define an object in the grammar or he can give partial information in one of three forms: a generic instance, an incomplete object, or an object to load. As part of completing the application definition, the system must actually instantiate the generic objects, complete incomplete objects by prompting the application specialist for values for each attribute, and physically load objects into the object base.

4.1.3.3 Preprocess Application The structured object base now contains only "complete", fully-instantiated application components. However, some critical data has not been specified. For example, the contents of a subsystem's import and export areas have not yet been identified. These areas are dependent upon the inputs and outputs, respectively, of the primitive objects which are controlled by that subsystem. Appropriate input-data and output-data for each primitive object have been identified during domain analysis and are available to the system in the technology base. Using this knowledge, the preprocessing module dynamically builds each subsystem's import and export areas, prompting the application specialist to indicate where the import data will be obtained when more than one subsystem produces the desired information.

4.1.3.4 Perform Semantic Checks Two levels of semantic checks exist in Architect. Architecture-oriented semantic checks ensure that the proposed application specification conforms to the composition requirements of the OCU model and that its behav-

ior can be successfully simulated (e.g., all components exist in the object base, application/subsystem update procedures directly reference only components which are part of the same application/subsystem, data input to one subsystem is produced as output by some subsystem in the application, etc.). Domain-specific semantic checks are knowledge-based, building on what is known about the domain, its objects, and previous applications created in that domain, to assist the application specialist in composing a meaningful and optimized application.

Meaningful architecture semantic checks can be performed on the application as a whole and also on its constituent subsystems. There are currently no meaningful semantic checks for primitive objects; the system assumes that primitive object class definitions and update procedures have been correctly constructed by a software engineer.

4.1.3.5 Edit Application If the application specialist decides an object instance is not exactly what was intended, he can edit the object. He can edit existing instances, add new objects, or delete objects. If the application specialist modifies the object base, he must also perform preprocessing and semantic checking on the entire object base to ensure the integrity of the data before simulating behavior or generating the specification.

The goal of domain independence has a large impact on how this module is designed. If certain domain knowledge is embedded in the source code, the code must be modified when the domain changes. If the code is completely independent, the interface may be more difficult to build and less user-friendly (the system can not give detailed prompts explaining what type of data is expected). In this case, domain independence is more important than friendly prompts.

4.1.3.6 Execute Application After the structured object base is fully populated and the semantic checks have uncovered no errors, the application's behavior can be simulated. This enables the application specialist to ascertain if the application, as specified, behaves as expected/desired. Behavior simulation is achieved by executing the application's **update procedure**, which consists of a series of calls to subordinate sub-

systems to execute their missions. Calling a subsystem to execute its mission invokes its **update procedure**. Subsystem update procedures consist of calls to subordinate subsystems/primitive objects, as well as **if** and **while** statements which allow conditional and iterative flows of control.

4.1.3.7 Save to Technology Base Since saved objects can be retrieved and parsed back into the object base, a function must exist to store objects in the object base into a file. The objects must be stored in the format required by the input procedure, that is, the format must adhere to the specifications of the domain-specific and architecture grammars. Saved application definitions can later be retrieved and loaded into the object base. Objects can be retrieved either through the grammar or through the interactive interface.

4.1.3.8 Generate Specification When the application specialist is satisfied that the specified application behaves as desired, he can generate its formal specification. The formal specification provides all the information necessary to directly code the application into an efficient production system. Indeed, the formal specification generated by this application composer is intended to be input to an automated code generation facility.

4.2 Preliminary Design of the Application Composer

Development of the full-scale application generator described in Chapter III will require several years of research and experimentation. This thesis effort focuses on one aspect of that project – the application composer and, more specifically, the Preprocess, Perform Semantic Checks, and Execute components. The implementation of these components is dependent on the choice of software architecture for the composed application. The Software Engineering Institute's Object Connection Update (OCU) model was selected as the software architecture for applications within Architect; the model can be easily formalized, appears to be applicable to a variety of domains, and has been used successfully by the SEI and other AFIT researchers (7, 40) for developing various simulators. This section briefly

reviews the OCU model, examines some of the goals/objectives of this implementation and describes the data structures used to support it.

4.2.1 Review of the OCU Model The OCU model describes a software architecture that is especially well-suited for developing software systems which can be "described as a set of subsystems" (24:17). It consists of three kinds of components: primitive objects, subsystems, and application executives.

4.2.1.1 Primitive Objects An object "models the behavior of a real-world or virtual component and maintains (its) state" (24:19). It is passive, activated only by an outside request to update its state. It is also very insular, aware only of its own internal data and is oblivious to other objects in the system including the sources of the external data it needs to update its own state. An object's internal data include (24:20-21):

- Input_Data: Information which is external to the object but is needed to update the object's state.
- Output_Data: Information which results from updating the object's state and which must be made available to other entities external to the object.
- Attributes: Descriptive characteristics of a particular instance of the object.
- Current_State: Data which defines the current state of the object.
- Coefficients: Data used in calculating the object's new state; can be modified to alter the object's behavior or state calculation.
- Constants: Information about the object which can not be changed.

Objects can be activated only through the following common, procedural interfaces (24:20):

- Update: Calculate the object's new state data (i.e., encapsulates the object's behaviorial description).
- Create: Allocate a new instance of the object.

- SetFunction: Change the function used to calculate the object's new state data and/or change the object's coefficients which can alter the behavior of the current update function.
- SetState: Change the object's state data directly, bypassing the update function.
- Destroy: Deallocate the object.

4.2.1.2 Subsystems A subsystem is an abstraction which represents a real-world subsystem. Some examples include the engine, electrical and airframe subsystems in a flight simulator. A subsystem consists of (24:18-19):

- A controller: "aggregates a set of objects (and possibly lower-level controllers) and manages the connections between them" (24:18). It is "the locus of a mission and the objects are services to carry out the mission" (24:18). Like primitive objects, the controller is passive and insular, aware only of the objects it connects and unaware of other subsystems in the application. A controller can be activated only by one of the following procedural interfaces:
  - Update: "Update object-connection network based on state data in the import area and provide new state data in the export area" (24:19). That is, it performs the subsystem's mission.
  - Stabilize: "Converge subsystem consistent with current scenario and make ready to operate... gets rid of transients" (24:19). In other words, it executes the subsystem's mission an appropriate number of times to let the data and/or algorithm converge to the proper value.
  - Initialize: "Activate supporting hardware... create objects and define object-connection network" (24:19); i.e., create the subsystem and the subsystems/objects which are controlled by it.
  - Configure: "Program the transfer characteristics of the controller and the objects" (24:19). Or, more simply, change some of the controlled objects' state data, coefficients and/or update functions.

- Destroy: Deallocate the subsystem, as well as the subsystems/objects which it controls.

- An import area: the focal point for external data needed by objects within the subsystem. It "makes the state data available to the objects by retrieving the data from other subsystems' export areas upon request for the data" (24:19).
- An export area: the focal point for data which is to be made available to external application components. "Data is placed in the export area where it is available to other subsystems' import areas" (24:19).
- Objects: the primitive objects (and/or lower-level subsystems) which are needed to accomplish the subsystem's mission.

4.2.1.3 Application Executive An executive is "an artifact of shared processor computing and provides the operating environment for the system" (42). It serves an a "activator" for the subsystems within the application, directing them to perform their missions as needed. Executives monitor interfaces to external entities and "manage time, the controllers (subsystems), and the application state to provide acceptable responses to stimuli" (42). This implies that the executive is the highest-level component in an application and encapsulates its mission.

4.2.2 Adapting the OCU Model for this Implementation The OCU model described above has proved to be very beneficial in separating "reaction strategy (or mission) from the providers of the strategic operations" (42). This separation is especially significant to application composition systems, as it allows virtually an infinite number of different missions to be created *without changing the system itself.* However, the OCU model is currently used only by computer professionals as an off-line tool for designing and coding new software systems; it is not used by end-users to dynamically compose applications and simulate their behaviors, as Architect will allow. The end-product of the model's usage to date has been fully-coded, unique software systems which satisfy a single (albeit complex) requirement. As such, various interactions among elements of the model (especially import/export areas and subsystems) have been "hardcoded", based on the specific

requirements of a single application. An application composition system must be flexible, capable of supporting a wide variety of applications in a wide range of domains and cannot rely on "hardcoded" interactions. Clearly, some additional consideration must be given to an implementation of the OCU model which precludes such "hardcoded" interactions.

4.2.2.1 Application Executive A fully-realized executive, capable of monitoring/controlling time and interfaces to external entities, was deemed too ambitious for this research effort, which is intended to demonstrate an application composition system "proof of concept;" enhancements to the application executive will be added in the future. In this implementation, an application consists of a collection of subsystems and an "executive subsystem." This "executive subsystem" is the forerunner of a fully-realized application executive and is treated as a specialized, high-level subsystem without import and export areas. Due to this lack of interfaces to external entities (i.e., input/output capabilities), there can be no external data; all data must be internal to the application.

4.2.2.2 Objects Architect expects that all to-be-combined components already exist; in fact, creating these components is one aspect of the system's Parse and Complete Application Components phases. Moreover, dynamic (that is, run-time) creation and deletion of objects greatly complicates the application composition process; it is extremely difficult to develop adequate semantic checks to ensure that such a proposed application can be successfully executed under all conditions. Therefore, the Create and Destroy OCU object procedural interfaces are not included in this implementation and are left for future research.

4.2.2.3 Subsystems As mentioned above, Architect expects all to-be-combined components to exist; this includes subsystems as well as objects. For reasons similar to those described above, the Initialize and Destroy OCU subsystem procedural interfaces are not implemented. In addition, due to some uncertainty about their utility, the Configure and Stabilize subsystem interfaces also are omitted from this implementation.

4.2.2.4 Import and Export Areas A subsystem's import area is the focus of external data needed by all objects aggregated by the subsystem and, conversely, the export

area is the focus of external data produced by those objects. Clearly, with dynamic composition, the application composition system must "know" what external data is needed by/produced by each object to allow proper construction of import and export areas; this should be quite easy to accomplish with an appropriately conducted domain analysis. But, given dynamically and properly constructed import and export areas, how should they be implemented and accessed to support the spirit of the OCU model?

There are several options for implementing import/export areas and their access functions. They include:

- Global Export Areas: All data which is to be made available externally to other primitive objects is stored in a single, global area, similar to a FORTRAN common area. When a primitive object requires external data, it is obtained directly from this common export area.

  This method, although easy to implement, violates the spirit of the OCU model, which clearly intends import and export areas to be localized within subsystems and to serve as the sole interface between primitive objects and the external data they require.

- Pre-Execution Data Retrieval or "Latching": Each subsystem has its own local import and export areas. Immediately before a subsystem is executed, all required external data is retrieved from appropriate export areas and copied into its import area; the subsystem can then provide the data from its own import area when a request is made for external data.

  This approach certainly conforms to the OCU model description and ensures that all imported values are temporally consistent. However, this method has a potentially serious drawback: external data produced during a subsystem's execution cannot be used in the same execution cycle by other primitive objects within the same subsystem. This may be (and most likely, is) too restrictive, considerably limiting the range of meaningful applications which can be created.

- Point-of-Use Retrieval: Like the option described above, this method provides each subsystem with its own local import and export areas. Unlike the earlier approach,

however, external data is retrieved as it is requested. This retrieval can be accomplished by the import area, which supplies the data directly to the requesting object with no involvement by other elements of the subsystem.

This approach also conforms to the OCU model description which allows "objects to access the import area's procedures directly" (24:19). It allows for very flexible subsystem construction, as external data produced by one object within the subsystem can be imported to another object in the same subsystem in a pipeline-type manner. Certain temporal restrictions on the data (i.e., a value must be exported before it can be used as an import) can be accommodated by judicious specification of the application/subsystem. This is the approach which will be used in this application.

4.3 Goals/Objectives for the Application Composer Implementation

The design of the application composer was influenced by the following goals/objectives:

- The implemented code should be domain-independent; domain-specific information should exist only in a technology base.
- An application should be expressible in domain-oriented terms as much as possible; computer software terminology and conventions should be kept to a minimum.
- An application definition should consist of an application executive, in addition to appropriate subsystem and primitive object components.
- A primitive object's mission is encapsulated within its update function. Subsystem and application executive missions should be specified by the user – any automatically generated mission would lack the flexibility required to adequately describe all possible missions for all possible domains. However, certain patterns of control may be identified for particular domains and may be applicable to a wide range of subsystem/executive missions. Identification and implementation of such patterns of control are left for future research.
- Alternative flows of control should be available for use in specifying application executive and subsystem missions – sequential flow of control places too great a limitation

on mission specification. Therefore, IF-THEN-ELSE and DO-WHILE constructs must be allowed. Further, variables must be allowed in if/while conditions to allow meaningful conditions to be specified.

- The application specialist should not be required to specify what external data is required by and produced by the primitive objects within his application. This information is already available to the application composer as a by-product of domain analysis and should reside in the technology base.
- Objects and subsystems should be unaware of the source of external data, where it is used, how it is stored, etc. This knowledge should be localized within import and export areas.
- External data needed by a primitive object should be retrieved as needed, not obtained en masse prior to subsystem execution. This allows a primitive object to use data just produced by another object within the same subsystem. With this retrieval scheme, there is no need to store retrieved data in the import area; it can be passed along directly to the requesting object.
- The application specialist should reference imported and exported data by name. This also pertains to identifiers in conditional expressions, as it has been established that they reference import/export items.
- The application specification should be complete before its behavior is simulated. Therefore, all import-to-export connections must be established before the application is executed. If more than one export datum can provide the information needed by an import, the application specialist must be prompted to select the appropriate one. If the application specialist truly does not care where the imported data comes from, he should be able specify that an appropriate, arbitrary source be used. These import-to-export connections are static; they are a fundamental aspect of the application specification and are not changed dynamically during application execution.

4.4 Conventions Used in this Implementation

The following conventions are used throughout the application composer:

4.4.1 Conventions For the Software Engineer

- All primitive objects have been correctly defined using the primitive object template described in Appendix A.
- All coded primitive object attribute/variable names are prefixed by the object's object class. For example: COUNTER-OBJ-COUNT – represents an attributed named COUNT which applies to objects of the class, COUNTER-OBJ. This scheme ensures that attribute/variable names are unique throughout the domain and presents a more domain-oriented (rather than programmer-oriented) "feel" to the application specification.
- All update function names begin with the object class. Example: COUNTER-OBJ-UPDATE1 is the name of the update function, UPDATE1, which is applicable to primitive objects of the class COUNTER-OBJ.

4.4.2 Conventions for the Application Specialist

- The application specialist, when referencing update function names and coefficients in **setfunction** statements and attributes in **setstate** statements, specifies only the unqualified name. For example, the application specialist would specify

```
setfunction counter1 update1
```

to set COUNTER1's update function to COUNTER-OBJ-UPDATE1 and

```
setstate counter1 (count, 2)
```

to set COUNTER1's COUNTER-OBJ-COUNT attribute to 2. This scheme allows the application specialist to use more domain-oriented (rather than programmer-oriented) terms and frees him from concern about object class names while preserving attribute name uniqueness throughout the domain.

4.5 Data Structures to Support this Implementation

The data structures chosen to represent the data in a software system and the programming language in which the code is written profoundly affect the system's implementation. For this implementation, we have selected the REFINE language and its grammar processing facility, DIALECT. This choice virtually dictates Architect's fundamental data structure: an abstract syntax tree. It also strongly encourages an object-oriented approach, as DIALECT relies heavily upon objects in its processing.

Figure 4.3 illustrates the hierarchy of object classes developed for this implementation. USER-OBJECT, the highest level object in the REFINE object class hierarchy, is the implied parent of each of the boxed object classes in the figure; this parent relationship has been omitted to allow all other meaningful relationships to be presented in one diagram. Figure 4.4 illustrates the attributes of each object class; for more detailed and technical information about these object class attributes, refer to the REFINE code in Appendix D.

4.6 Summary

This chapter presented an overview of the high-level design of Architect, the application composition system which was introduced in Chapter 3.5. Architect is heavily dependent on the software architecture used to produce its compositions; therefore, the selected architecture (the OCU model) was briefly discussed, as were various adaptations which were made for this implementation. In addition, the goals/objectives of this application composer implementation were enumerated. Lastly, several conventions and data structures, which are used throughout the system, were presented.

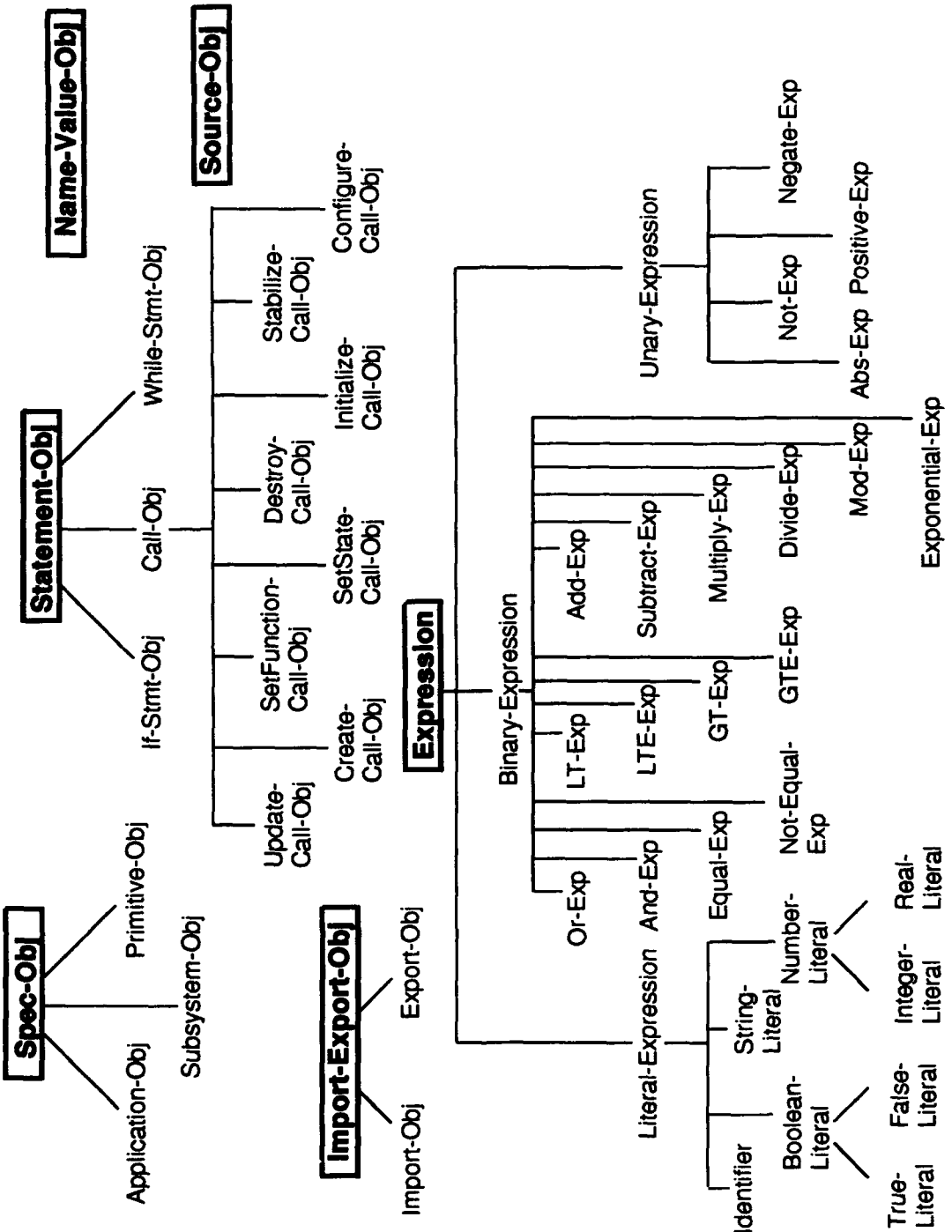

Figure 4.3. REFINE Object Class Hierarchy

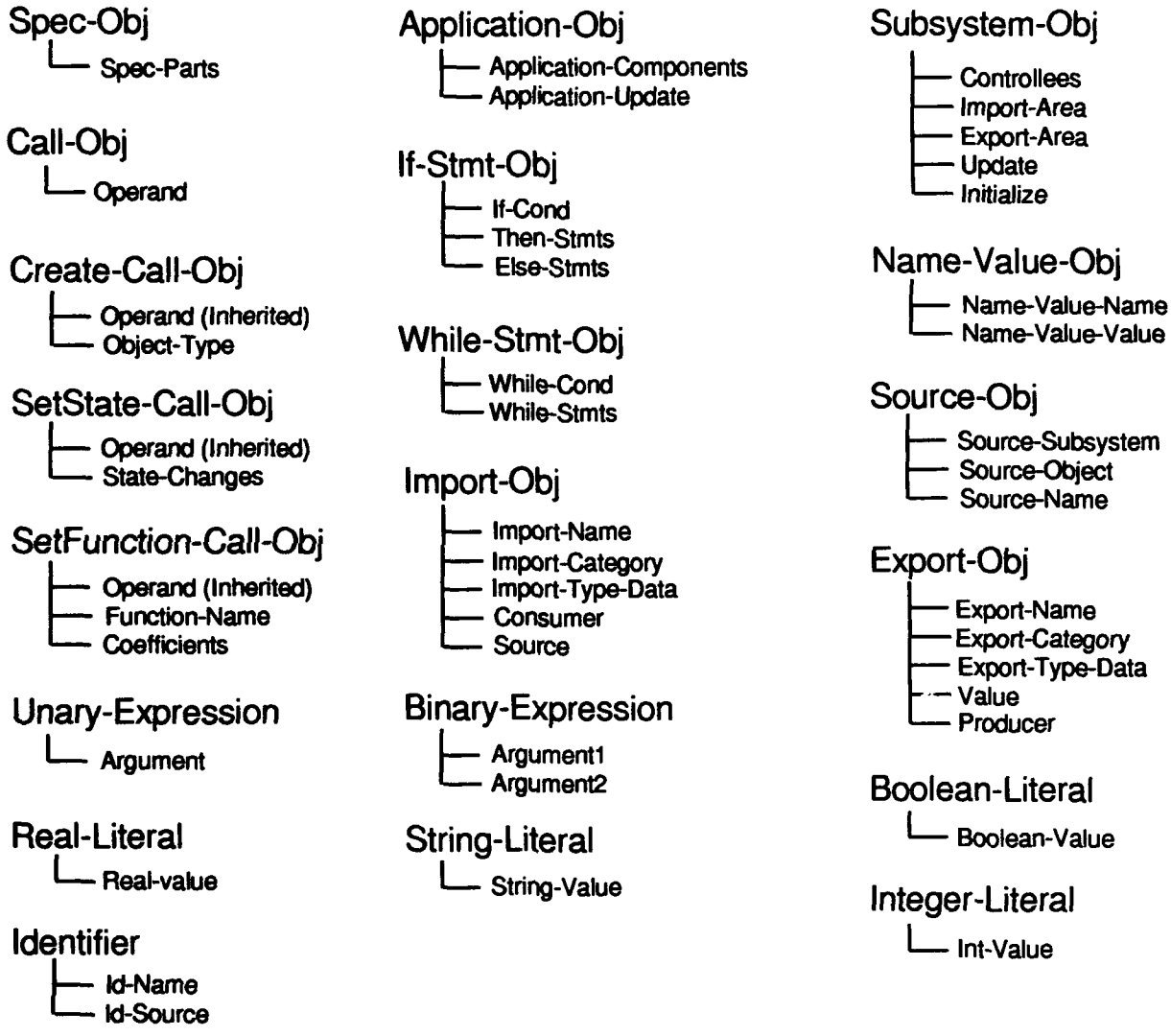

Figure 4.4. Object Class Attribute Maps

V. *Detailed Software Design*

This chapter presents a detailed design of the Preprocess, Semantic Checks and Execute portions of Architect, the application composition system introduced in Section 3.5. It elaborates on the high-level design of these major functions discussed in Section 4.1, meets the implementation's goals/objectives (reference Section 4.3), and conforms to the conventions identified in Section 4.4.

5.1 Preprocess the Application

After the application is entirely defined, the structured object base contains "complete", fully-instantiated components (from the application specialist's viewpoint). However, some critical data needed for further processing may not yet be specified; preprocessing the application obtains such data from available system knowledge, making it accessible in a more usable form and, thus, "completing" the specification. Two examples of such critical, as-yet-unavailable data are subsystem import areas/export areas and the source of identifiers used in `if` and `while` statements in subsystem update procedures.

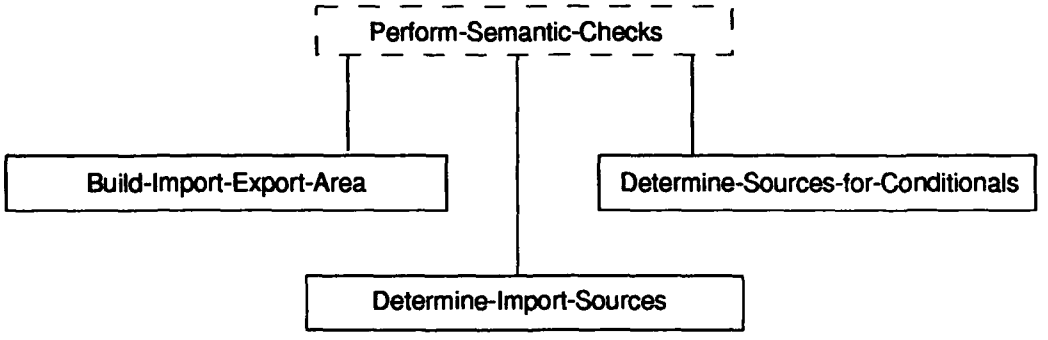

Figure 5.1. Preprocess Application

A structure chart representing the preprocessing activity is presented in Figure 5.1. Preprocessing is not an inherent requirement for all application composers; it evolved as a requirement for this application composer due to our selection of the OCU architectural model as its basis. Because it may not be required for a different application composer implementation, preprocessing should be transparent to the application specialist; that is,

he should be virtually unaware of its existence. Therefore, although it is conceptually a stand-alone component of this implementation, preprocessing has been incorporated into the semantic check component.

5.1.1 Building Import and Export Areas The contents of an import area depend upon the external data (input-data) needed by all the primitive objects which are controlled by that subsystem; likewise, a subsystem's export area depends upon the data (output-data) produced by all its primitive objects which must be made available to other subsystems/objects. In keeping with our goal of constraining application specification to domain-oriented rather than programmer-oriented terms, an application specialist should not be required to specify the contents of import and export areas of the subsystems in his application. Moreover, input-data and output-data for each class (or type) of primitive object are available in the technology base as a consequence of domain analysis. Therefore, the appropriate data for each import area and export area can be generated automatically, given a list of the primitive objects which comprise the subsystem (the `controls` clause serves as such a list). This automatic generation of import and export areas is accomplished via preprocessing.

Figure 5.2 illustrates this process of automatically "building" the import and export areas. The upper portion of the figure represents a subsystem *before* preprocessing is accomplished. Note that its objects "contain" input-data and output-data but that the import and export areas are empty (input-data can be distinguished by the partitioning of its rightmost segment into three horizontal parts; this will be explained later). Preprocessing the subsystem transforms it into the representation at the bottom of the figure. Note that all input-data for all objects within the subsystem have been copied to the import area and all output-data to the export area. Also, note that the consumer object name and producer object name have been added to import items and export items, respectively (in the box, second from the right).

5.1.1.1 BUILD-IMPORT-EXPORT-AREA Each subsystem in an application definition is examined. For each input data item for each primitive object controlled

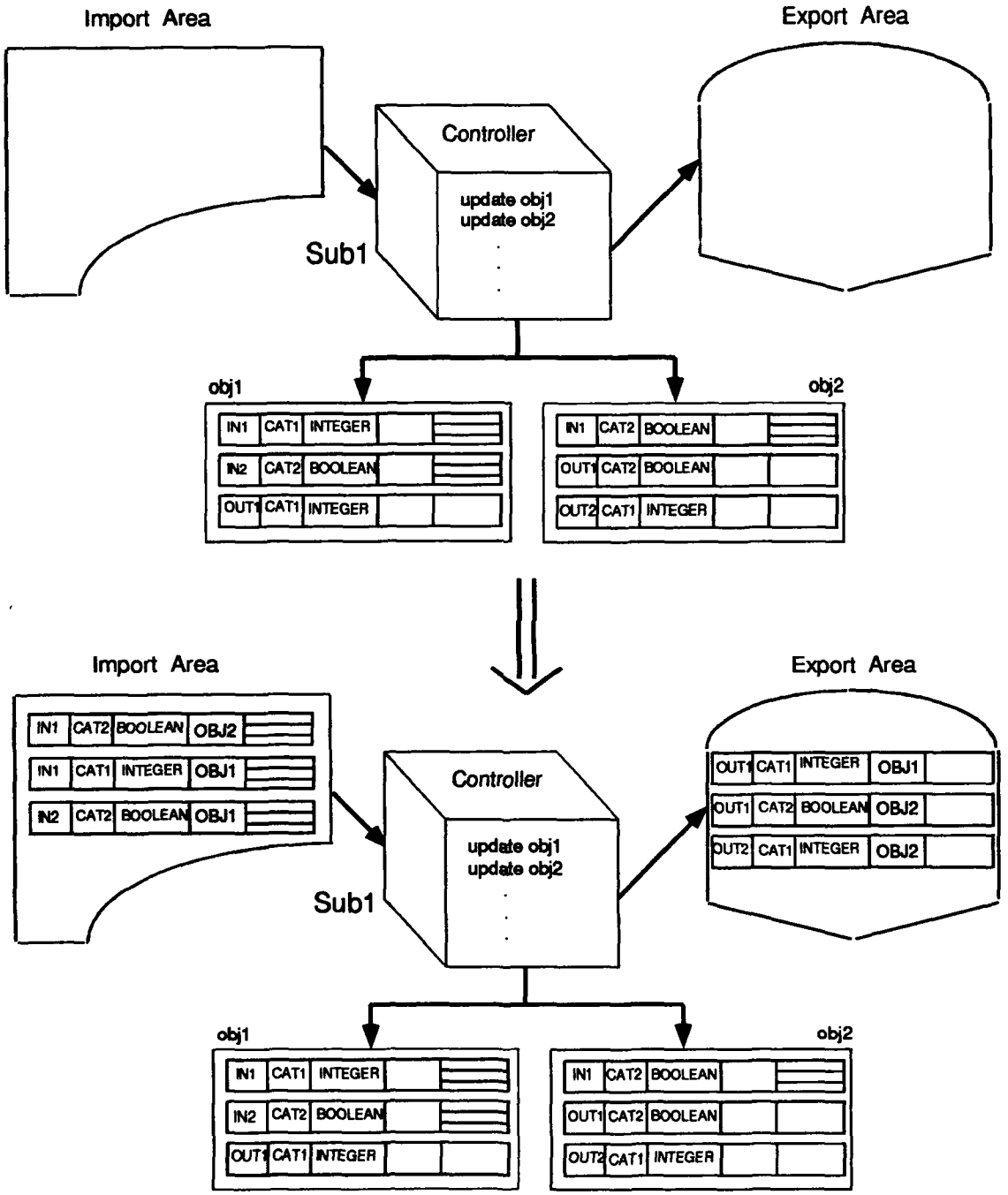

Figure 5.2. Build Import/Export Areas

by the subsystem, an entry is made in the subsystem's import area; likewise with output data and the export area.

5.1.2 Determining the Source for Imports Merely specifying the contents of import/export areas is inadequate to complete the application's specification with respect to external data requirements. Under the OCU model, a primitive object's request for a piece of external data (i.e., an import area entry) is satisfied by retrieving the appropriate data from an export area in some subsystem within the application. To complete the specification, these connections between each item in an import area and the export item which is to provide its data must be made. In other words, the source of the data for each import item must be specified. This, too, is accomplished via preprocessing.

When an import can be satisfied by only one export item (see Section 5.1.3), its source can be automatically determined without user involvement. This is illustrated in Figure 5.3. IN1, which is needed by OBJ2, and IN2, which is required by OBJ1, are both of type CAT2; only OUT1, which is produced by OBJ2, can provide this type of data. Therefore, OUT1 *must* be the source for these imports. The subsystem name, producing object name, and output-data name of the source export item appear from top to bottom in the import item's rightmost segment.

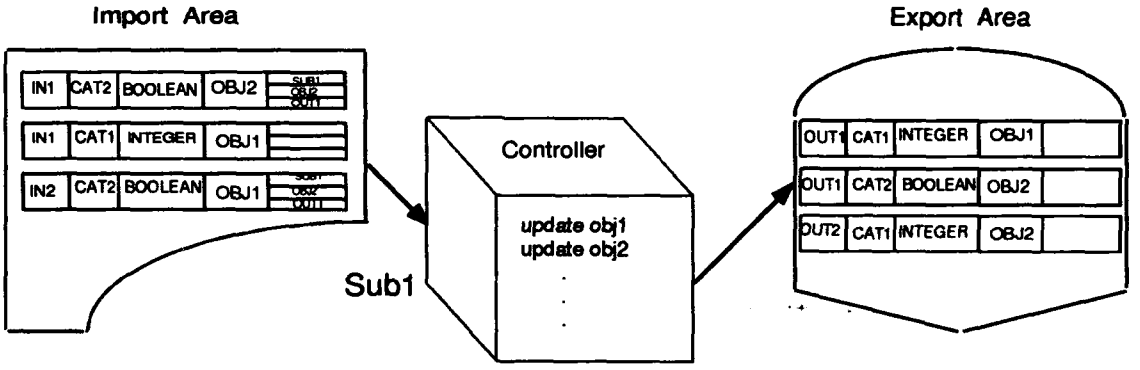

Figure 5.3. Determine Import Sources – Part 1

However, the source of an import which can be satisfied by more than one export item can not be determined automatically; the application specialist must indicate which

potential source should be used. Figure 5.4 represents the state of the import area after the following source determination dialogue has taken place:

```
More than one export can provide the data for IN1
                    which is used by object OBJ1
                                in subsystem SUB1
Choose the export item (subsystem and component)
  that you wish to be the source of this data:
     1> subsystem "SUB1" component "OBJ1" name "OUT1"
     2> subsystem "SUB1" component "OBJ2" name "OUT2"
Enter the number corresponding to the source you want to use
2
```

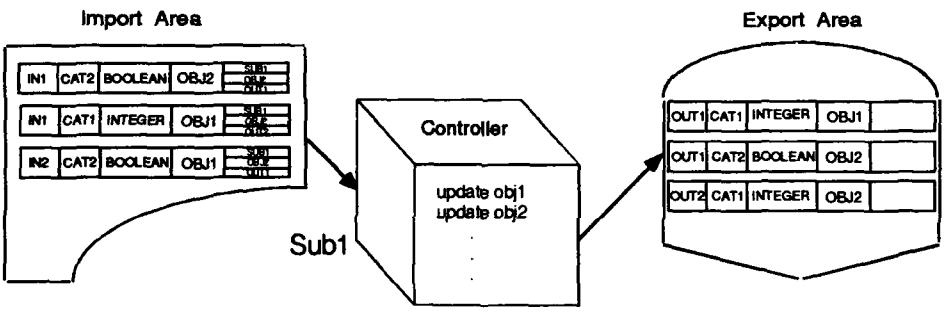

Figure 5.4. Determine Import Sources – Part 2

Note: these import-to-export connections can be made only after all import/export areas have been constructed to ensure that all exports are considered as possible import sources.

5.1.2.1 DETERMINE-IMPORT-SOURCES Each item in the import area of each subsystem in the application definition is examined. If a source has not yet been specified for that import, the export areas of all subsystems in the application are searched for export items which could provide the needed data. Only export items which are of the same data category as the import item can be considered as potential sources. If only one export item produces data of the proper category to satisfy an import item, it is automatically identified as the source. If more than one export item could provide the

required data, the application specialist is prompted to select the appropriate source from a list of possible choices, which includes a "use an arbitrary one" option (to be used if the application specialist doesn't care where the data comes from; it is anticipated that this will be rarely used). If a source has been previously specified for the import item, it should be displayed to the application specialist, who may select a different source at this time, if desired.

5.1.3 Import/Export Considerations The following questions or considerations arose during different phases of this design and its implementation. When feasible, a rapid prototype was constructed to answer these questions and to test various alternatives. There were two underlying principles which dictated the choices made in answering these questions: allow maximum flexibility for application specification and free the application specialist from implementation details as much as possible.

1. How are the external data needed by and produced by a primitive object known? One aspect of domain analysis is to determine what external information (INPUT-DATA) is needed to adequately process (update) each class of primitive object and what information must be made available externally to other objects (OUTPUT-DATA). Implicit in determining each required INPUT-DATA and OUTPUT-DATA is the identification of its name, category, and basic data type. Because INPUT-DATA and OUTPUT-DATA are the same for each object instance in a particular object class, this information can be pre-stored in the technology base and is therefore available to be incorporated automatically into each newly created object of that class.
2. Does each instance of a primitive object import the same value for a given variable name? It seems likely that different object instances, which require external data with the same name, may actually wish to obtain that data from different sources. Therefore, only one entry in the import area for each required external variable name is insufficient; the name of the requesting object must be maintained as well to ensure that each primitive object's external data request accesses the correct import item. When a piece of external data is required by a primitive object update function,

both the requested variable name and requesting object's name (consumer) must be matched to ascertain the correct import area entry to be used in obtaining the appropriate requested data.

3. Does a subsystem export only one value per variable name? Just as it was likely that different object instances may obtain INPUT-DATA with the same name from different sources, it is also likely that more than one object in a subsystem may produce external data with the same variable name. Therefore, one entry in the export area per variable name is insufficient; the producing object's name (producer) must be maintained as well as the variable name and its associated value. When data is to be stored into the export area, both variable name and producer name must be matched to ensure that the correct export item is used.

4. How does an object obtain the external data that it requires? External data is requested via a GET-IMPORT call whose parameters include the name of the data to be obtained as well as the names of the requesting primitive object and its subsystem. GET-IMPORT finds the appropriate import item within the subsystem's import area which corresponds to this request and uses its previously specified source information to directly obtain the needed data.

5. What is "source information?" It represents a connection between an import item and the export item within the application which supplies the data to be imported. Source information consists of the name of the subsystem in whose export area the "source" export item can be found, the name of the primitive object which produces that data, and the name of the data itself. This information uniquely describes the export item from which the imported data is to be retrieved.

6. How does an export item qualify as the source of an import item? Each piece of external data (both imports and exports) has a name, a data category and a data type. Together with the subsystem name and the name of the consumer/producer object, the name provides a means to uniquely identify an import or export item. The data type indicates the item's primitive data type (integer, real, boolean, string or symbol); this is used when dealing with identifiers in conditional expressions. The data category indicates the class of the data, in domain-oriented terms. For

example, an import might be expecting a real number (data type) but its category may be water-temperature or air-pressure or interest-rate, etc., depending on the requirements of the domain. Only those export items which are of the same data category can be potential sources for the import. This scheme, in effect, creates user-defined, domain-dependent data subtypes (very similar to Ada subtypes) which serve to constrain the possible data choices. In the previous example, although export items representing water-temperature, air-pressure and interest-rate may all be real numbers, only the water-temperature export can serve as the source for a water-temperature import. In the current implementation, data category and data type are related; all imports/exports of a given category are assumed to be of the same underlying data type. If this later becomes too restrictive, conversion functions may be required to allow disparate data types (e.g., integer and real) to be used interchangeably within a data category.

7. What if more than one subsystem and/or object produces external data which qualifies as the source of an import item? If more than one subsystem/object produces data which can satisfy the request, the application specialist is prompted to indicate which data should be used to satisfy the request rather than allowing the system to choose arbitrarily. However, after the appropriate source is specified for a particular request, the system should not prompt the user again if the same data is later requested by the same object; it should "remember" which subsystem/object produced the requested data and access it directly. To allow the system to "remember" such information, source subsystem name and source object name are stored in the import area in addition to the source's variable name.

8. What if the application specialist doesn't care where the requested data is obtained? If only one export item can provide the requested data, it obviously should be used as the data source. If more than one export item may produce the data, the system should allow the application specialist to use an arbitrary source if *any* data of the proper category will suffice. The "arbitrary source" option is provided in addition to the specific subsystem/object choices discussed previously. If the application specialist selects the arbitrary source option, all future requests for the same data by

the same subsystem will be satisfied by the selection of an arbitrary source which satisfies the data category requirement.

9. When an application is edited, what happens to the import and export areas of its subsystems? The edit process can change any/all of the components already in the application and can also add/delete objects to/from the application. Adding/deleting objects from subsystems will likely change import and/or export areas; therefore, preprocessing must be reaccomplished after each edit step. To ensure that preprocessing is accomplished, a new-data flag is set each time the edit process is initiated. If the new-data flag is set, preprocessing and semantic checks must be reaccomplished on the entire application specification before its behavior can be simulated. During preprocessing, import items are added to the subsystem's import area if there are currently no import items for a primitive object and likewise for the export area; this ensures that import/export areas are consistent for *added* subsystem components. A "clean-up" operation is also conducted. Any import item used by a primitive object which is no longer part of the subsystem is removed from the import area, and likewise for the export area; this ensures that import/export areas are consistent for *deleted* subsystem components.

5.1.4 Determining the Source of Variables in Conditions `If` and `while` statements within a subsystem update procedure provide the necessary flexibility which enables an application specialist to precisely specify the required mission for any subsystem. Each of these statements includes a conditional expression, the result of whose evaluation determines how the statement will be executed. In general, meaningful conditional expressions include some variable data whose value changes during the current execution cycle or from one execution cycle to the next. During specification, the application specification merely provides a name or identifier for this variable data. To complete the specification, this identifier must be associated with the data to which it refers. The only data directly available to the subsystem's **update procedure** resides in its import and export areas; therefore, the identifier must be found in one of these areas.

5.1.4.1 DETERMINE-SOURCES-FOR-CONDITIONALS Each identifier in each subsystem update procedure in an application definition is examined. If the identifier has not yet been associated with its corresponding data, the subsystem's import and export areas are searched to find potential association candidates. If there is only one possible candidate, use it as the identifier's source; if there are multiple potential candidates, present the list to the application specialist who must select the appropriate one. No possible candidates indicates a specification error. If the identifier/data association has already been accomplished, the application specialist is notified, presented with the current association, and may select a different association, if desired.

5.1.5 Considerations for Variables in Conditional Expressions As with the import/export area considerations, these questions concerning variables or identifiers in IF and WHILE conditional expressions had to be resolved before the design and implementation could be realized.

1. What variable data (identifiers) can be used in the conditional expressions of `if` and `while` statements in subsystem **update procedures**? Identifiers must be recognized by and available to the subsystem in whose **update procedure** they appear. The OCU model does not provide a mechanism for a subsystem to directly access any of its primitive objects' current-state data. The only data available to a subsystem resides in its import and/or export areas. Therefore, an identifier can only be associated with an import or export item. The application specialist specifies the identifier by name; if an import or export item has the same name, it is a candidate for association. A specification error occurs if there are no candidates within the subsystem.
2. How do you know that conditional expressions involving identifiers are valid (i.e., that you aren't trying to compare apples and oranges)? Each import and export item has a "data-type" which corresponds to a REFINE primitive data type (e.g., integer, real, string, etc). It is used to ensure that conditional expressions can be evaluated in a meaningful way.
3. If several imports and/or exports within a subsystem have the same name, how does the system (and application specialist) determine which one applies to each variable

used in the conditional? As with determining the sources for imports, Architect first finds all possible candidates (i.e., imports and exports with the same name as the conditional variable). If only one item meets that criteria, it is obviously *the* intended source and Architect automatically makes the connection. If, however, there are multiple possible sources, the system prompts the user to select the desired source from the a list of alternatives. The user may by-pass this source determination dialogue by "qualifying" the variable name(s) in the conditional. This is achieved by prefacing the variable name with the object which produces/consumes that item (this assumes that imports and exports for the same object type do not share common names). For example, if a variable in a condition refers to the import item named IN1 which is consumed by OBJ1, the user would specify the qualification as `OBJ1.IN1`. After sources for conditionals have been determined, it is impossible to differentiate between qualified and unqualified variables.

5.2 Perform Semantic Checks

Two levels of semantic checks exist in Architect. Architecture-oriented semantic checks ensure that the proposed application specification conforms to the composition requirements of the OCU model and that its behavior can be successfully simulated (e.g., all components exist in the object base, application/subsystem update procedures directly reference only components which are part of the same application/subsystem, data input to one subsystem is produced as output by some subsystem in the application, etc.). Domain-specific semantic checks are knowledge-based, building on what is known about the domain, its objects, and previous applications created in that domain, to assist the application specialist in composing a meaningful and optimized application. This research effort currently includes only architecture-oriented semantic checks; domain-specific semantic checks are presently being investigated by Captain Mark Gerken, an AFIT doctoral student.

After preprocessing has been completed, architecture semantic checks are performed. These semantic checks embody the constraints imposed on the composition of applications within the framework of the selected architecture model (in this case, the OCU model). They are derived solely from the structure of the architecture model and are domain-

independent. Each semantic check is encapsulated into a REFINE function; these functions are executed via an appropriate sequence of function calls. Domain-specific semantic checks are completed after the proposed application successfully passes the architecture semantic checks.

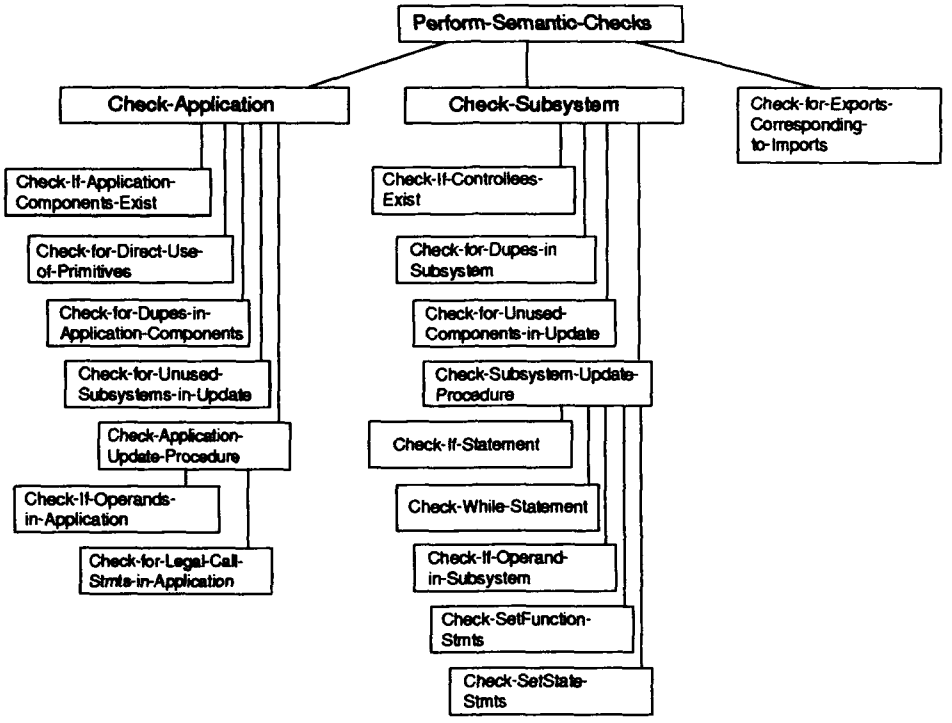

Figure 5.5. Semantic Checks

5.2.1 Architecture Semantic Checks Meaningful architecture semantic checks can be performed on the entire application and its constituent subsystems. There are currently no meaningful semantic checks for primitive objects; Architect assumes that primitive object class definitions and update procedures have been correctly constructed by the software engineer. A structure chart for processing semantic checks appears in Figure 5.5.

Architecture semantic errors describe conditions that preclude successful behavior simulation of the application or present an unacceptable inconsistency which can not be successfully resolved without human intervention. Warnings, on the other hand, represent apparent inconsistencies in an application specification which may actually presage a com-

position error but do not preclude behavior simulation. However, they should be carefully considered by the application specialist before proceeding. The semantic checks in this implementation generate either errors or warnings, as appropriate.

5.2.1.1 Global Application Specification Semantic Checks The following checks are conducted on the entire proposed application specified by the application specialist.

1. Each application specification must contain one and only one application executive object. An example of a very simple application object is:

```
application application1 is
    controls:  subsystem1, subsystem2
    update procedure:
        update subsystem1
        update subsystem2
```

2. An instance of a primitive object can be part of only one subsystem within an application. This restriction is necessary to ensure the integrity of each subsystem. Only activities within the subsystem (and their inputs) affect the operation of the subsystem; the states of the objects which comprise a subsystem are changed only by executing the subsystem's **update procedure**. An object instance which is part of two or more subsystems could cause spurious results because its state would be determined by multiple subsystems.

3. Each application specification must contain all those (and only those) components needed to compose the application. Unused subsystems (not included in an application's **controls** clause nor incorporated into any other subsystem) and primitive objects (not included in any subsystem's **controls** clause) may indicate an oversight by the application specialist - perhaps he actually intended to use them in the application but forgot to do so. If this anomalous condition is discovered, a warning is issued.

5.2.1.2 Application Executive Semantic Checks The following checks are conducted on each application executive found in the proposed application specification. In

theory, only one application executive exists per specification. However, in an effort to provide the application specialist diagnostic information on all anomalies at the earliest possible phase, all application executives are checked in this manner.

1. CHECK-IF-APPLICATION-COMPONENTS-EXIST: Subsystems listed in the **controls** clause must already exist (i.e., be specified in the application definition). Note: although the OCU model allows for dynamic creation of subsystems via an initialize operation, there are several unaddressed issues regarding dynamic initialization, and it has not yet been included in this implementation.
2. CHECK-FOR-DIRECT-USE-OF-PRIMITIVES: Only subsystems and subsystem procedural interfaces may be referenced in an application's **controls** clause and **update procedure**; direct reference to primitive objects is not allowed. It is unclear whether the OCU model itself allows arbitrary primitive objects to be included directly in an application executive; certainly, such statements as "the OCU has most effectively been applied when the software system under development can be described as a set of subsystems" (24:17) imply the contrary. From an implementation viewpoint, several unresolved issues regarding application executives (e.g., do they have facilities similar to import/export areas and, if so, how should they work? Is it *desirable* to have special, primitive objects for executives?) currently preclude the direct use primitive objects.
3. CHECK-APPLICATION-UPDATE-PROCEDURE: At this time, only call statements (i.e., the OCU subsystem **update** procedural interface) are valid in the application **update procedure**. **If** and **while** statements are not allowed in this very simple application executive implementation because no appropriate data is available from which to construct (and evaluate) meaningful conditions. Recall that condition variables must be accessable at the level at which they are specified and that there is no provision in the OCU model to query a subordinate object about its state data. This restriction was overcome for subsystems by associating condition variables with import/export items. At the present time, no import or export areas are associated with applications.

- CHECK-IF-OPERAND-IN-APPLICATION Only subsystems included in the application's **controls** clause may be referenced in its **update procedure**. This constraint requires the application specialist to carefully consider which subsystems are needed in his application *before* thinking about how they are to be composed. It also allows Architect to compare the **controls** entries against the operands in the **update procedure** as a consistency "double check."
- CHECK-FOR-LEGAL-CALL-STATEMENTS: Currently, only the update interface is implemented for subsystems; too many unresolved issues remain to implement the remaining subsystem interfaces (initialize, stabilize, configure and destroy) at this time. However, in anticipation that future research efforts will resolve these issues, the OCU grammar accepts these interfaces as valid keywords; this REFINE function ensures that no attempt is made to actually execute them.

4. CHECK-FOR-DUPES-IN-APPLICATION-COMPONENTS: Is an application component listed more than once in the **controls** clause? Such a duplication may actually have been a typographical error and not what the application specialist intended. A warning is generated.
5. CHECK-FOR-UNUSED-SUBSYSTEMS-IN-UPDATE: Are there any subsystems listed in the **controls** clause which are not used as operands in the application's **update procedure**? Such an omission may have been an oversight and additional statement(s) should have been included in the **update procedure** to execute these subsystems. Or, the subsystem in question may have been included in the **controls** clause erroneously. A warning is generated.

5.2.1.3 Subsystem Semantic Checks The following checks are conducted on each subsystem found in the proposed application specification.

1. CHECK-IF-CONTROLLEES-EXIST: Components (subsystems and primitve objects) listed in the **controls** clause must already exist. Note: although the OCU model does allow for dynamic creation of subsystems via an initialize operation and

primitive objects via the create interface, there are several unaddressed issues regarding such dynamic creation; therefore, the create and initialize interfaces have not yet been incorporated into this implementation.

2. CHECK-SUBSYSTEM-UPDATE-PROCEDURE: All of the statements within the subsystem's **update procedure** must be examined to ensure that only legal actions are specified. Various semantic checks are executed, depending on the type of statement encountered.

   - CHECK-IF-STATEMENT: **If** conditions must be valid – each condition must be reduceable to a boolean expression, all identifiers referenced must be available in the subsystem's import and/or export area, data types within the condition must be compatible with each other and with the operation specified (e.g., arithmetic operations can not be performed on data of type STRING). In addition, all the statements within the **if** statement (both the **then** and **else** clauses) must be valid.

   - CHECK-WHILE-STATEMENT: **While** conditions must conform to the same restrictions as **if** conditions; during semantic checking, no distinction is made between **if** and **while** conditions. In addition, all statements within the **while** loop must be valid.

   - CHECK-IF-OPERAND-IN-SUBSYSTEM: Controllers (i.e., subsystem **update procedures**) may access only those components (subordinate subsystems and primitive objects) which are part of the subsystem (that is, included in its **controls** clause). Currently, the application specialist indicates which components are part of the subsystem and then specifies the update procedure for the subsystem. This semantic check is actually a consistency check to ensure the **controls** clause and **update procedure** are compatible. This check is applied to all call statements.

     – Should the application specialist be required to explicitly specify the components which comprise a subsystem? Doing so allows the above described consistency check between the controllees and the update procedure to be

performed as a "double check"; this check has proved helpful during testing to keep the application specification "on-track." Or, should the aggregation be implicit based on the components included in the update procedure? This approach would make it tedious to check which components comprise the subsystem and does not allow the "double check" mentioned.

- CHECK-SETFUNCTION-STMT: The function name and coefficients specified in a **setfunction** statement must be valid. That is, the function name must identify an existing function and the parameters of that function must include a primitive object of the same type as the object specified as the operand of the **setfunction** statement. Each coefficient specified must be valid for the statement's operand.

  - Should the application specialist be required to specify the complete function name (which may be quite long and code-like, especially if it is prefaced by its associated object class type as Architect requires)? Or, should the application specialist be required to identify just the latter portion of the name, the distinguishing part? Keep in mind, the entire function name can be generated by Architect easily by prepending the object class of the statement's operand to this distinguishing name. In an effort to keep Architect very domain-oriented rather than programming-oriented, we have opted to take the latter approach, constructing the entire function name given just its distinguishing portion.

- CHECK-SETSTATE-STMT: State variables and their new values specified in a **setstate** statement must be valid. That is, the variable name must identify an actual attribute of a primitive object of the same type as the object specified as the operand of statement and the new value must be of the same data type as required for that attribute. Note: as with the function names in **setfunction** statements and for the same reason, the application specialist specifies just the distinguishing part of each attribute name; Architect automatically generates the complete name by prepending the object class name to this distinguishing part.

3. CHECK-FOR-EXPORTS-CORRESPONDING-TO-IMPORTS: Input data required by one subsystem must be produced as output data by another subsystem within the application. If no subsystem exports data which can serve as the source for the input-data, the execution simulation can not be processed correctly.

    - This check ensures that, for each import item, at least one export item can be found that produces suitable data, i.e., that some subsystem is potentially capable of producing the needed data. It can not, however, assure that the correct data is actually available when needed during the behavior simulation. For example, assume that SUBSYSTEM1 produces an export named OUTPUT1 of category TYPE1 and SUBSYSTEM2 requires an import named DATA1 of category TYPE1. If SUBSYSTEM2 is executed before SUBSYSTEM1, OUTPUT1 will not actually be available when needed. Further, using the above example, even if SUBSYSTEM1 is executed before SUBSYSTEM2, there is no guarantee that a valid OUTPUT1 will be available to SUBSYSTEM2; it could be produced in a conditional statement whose condition is not met during execution.

4. CHECK-FOR-DUPES-IN-SUBSYSTEM: Is a subsystem component listed more than once in the **controls** clause? Such a duplication may actually have been a typographical error and not what the application specialist intended. A warning is generated.

5. CHECK-FOR-UNUSED-COMPONENTS-IN-UPDATE: Are there any subsystems or primitive objects listed in the **controls** clause which are not used as operands in the subsystem **update procedure**? Such an omission may have been an oversight and other statements need to be added to the **update procedure**. Or, the subsystem/primitive object may have been included erroneously in the **controls** clause. A warning is generated.

5.3 Simulate Execution

After the proposed application specification passes all semantic checks, it is fully specified and there are no obvious specification errors which would preclude behavior simulation. Ideally, this behavior simulation or execution demonstrates to the application

specialist that the application he has specified does, indeed, behave as intended. If the application does not behave as intended, the application specialist may edit the offending application specification to "correct" it or may begin again with another complete application definition.

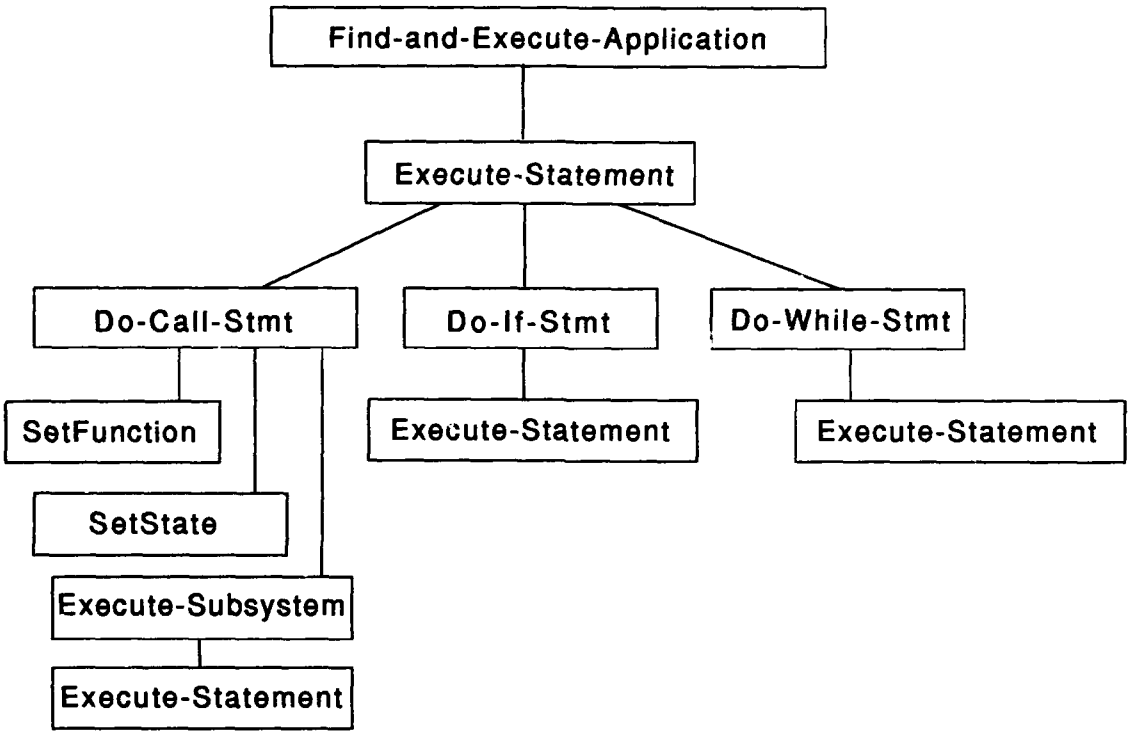

Figure 5.6. Execute Application

Figure 5.6 presents a structure chart illustrating the behavior simulation process. Application execution is accomplished by invoking EXECUTE-STATEMENT for each statement in the application update procedure. Semantic checks have already verified that all the statements within this **update procedure** are call statements (i.e., updates) whose operands are subsystems – due to the simplicity of the currently implemented application executive, **if** and **while** statements are not allowed in the application executive's **update procedure**.

The mission of a subsystem is performed by EXECUTE-SUBSYSTEM, which calls EXECUTE-STATEMENT for each statement in the subsystem's update procedure. EXECUTE-

STATEMENT performs the appropriate action for the type of statement (e.g. call, **if**, **while**) encountered.

5.3.1 Call statements Call statements are processed by DO-CALL-STATEMENT, which performs the appropriate function based on the type of call requested.

- **update** If the operand of the **update** is a subsystem, EXECUTE-SUBSYSTEM is called to perform its mission. If the operand of the statement is a primitive object, the current value of UPDATE-FUNCTION is retrieved and serves as the first parameter to the lisp FUNCALL function. The value of this first parameter determines which REFINE function is to be invoked. Thus, FUNCALL provides a mechanism which allows dynamic, run-time determination of the function to be called and allows the behavior simulation code to remain domain-independent.

  Figure 5.7 depicts the syntax and semantics of an object update call. The syntax for the call is presented, followed by a sample statement. The rectangle represents the statement's operand, **widget1**. The current value of **widget1**'s **update-function** (in this case, **widget-obj-update1**) becomes the first parameter to the lisp FUNCALL function; its succeeding parameters serve as parameters to the function invoked by FUNCALL (in this case, **widget-obj-update1**).

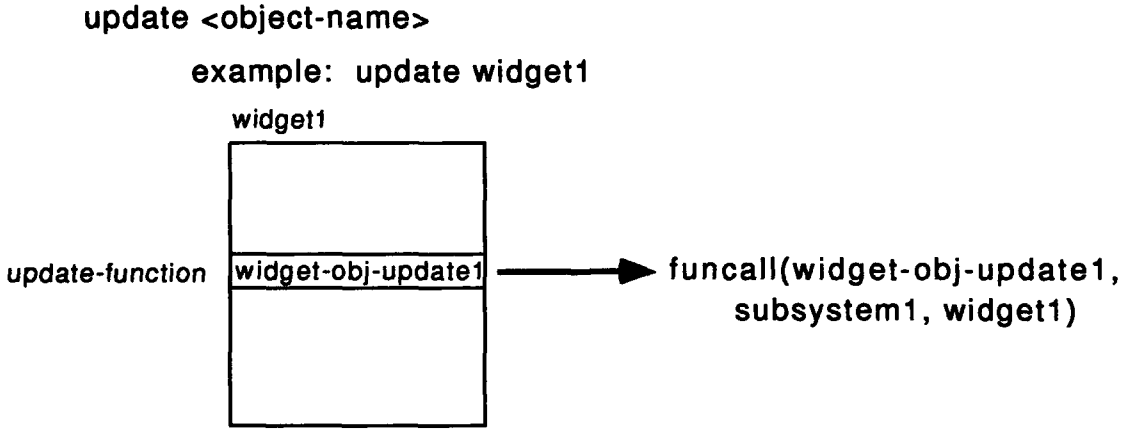

Figure 5.7. Primitive Object Update Execution

- **setfunction** The **update-function** of the statement's operand is replaced by the function name specified in the statement. In addition to changing the value in its operand's UPDATE-FUNCTION, the **setfunction** statement also provides the means to change the current value of any or all of its operand's coefficients. This is pictured in Figure 5.8 which shows the statement's syntax and a sample statement, as well as the effect of executing that sample statement. Because one of our goals is to make application specification a domain-oriented rather than programmer-oriented process, the application specialist is not required to (in fact, should not) specify the entire update function name; the entire name is constructed by prepending the name of the primitive object's object class to the function name specified by the application specialist. Note: the semantic checks, which must be successfully performed before the behavior simulation can begin, assure that the specified function name and coefficient/value pairs are legal.

setfunction <object-name> <new-update-function-name> (coefficient$_1$, new-value$_1$)*

example: setfunction widget1 update2 (coef1, 15) (coef2, 4.7)

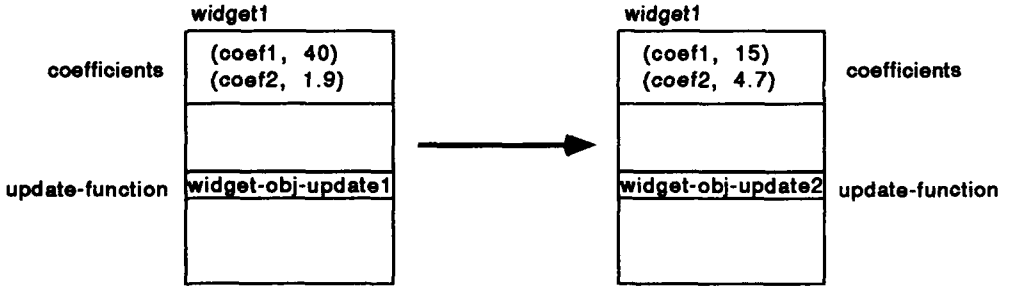

Figure 5.8. SetFunction Execution

- **setstate** Setstate enables the application specialist to directly change the value of any of its operand's attributes (unlike the OCU model, this implementation does not make a distinction between "attributes," "state data," and "constants," all of which may be changed via **setstate**). The syntax, a sample statement and the effect of executing that statement are depicted in Figure 5.9. As with the update function name, the entire attribute name is automatically generated by prepending

the operand's object class to the attribute names specified by the application specialist. Again, note: the semantic checks, which must be successfully performed before behavior simulation can begin, assure that the specified attribute names and their new values are legal.

setstate <object-name> (attribute$_1$, new-value$_1$)*

example: setstate widget1 (a, 14) (b, 5.5)

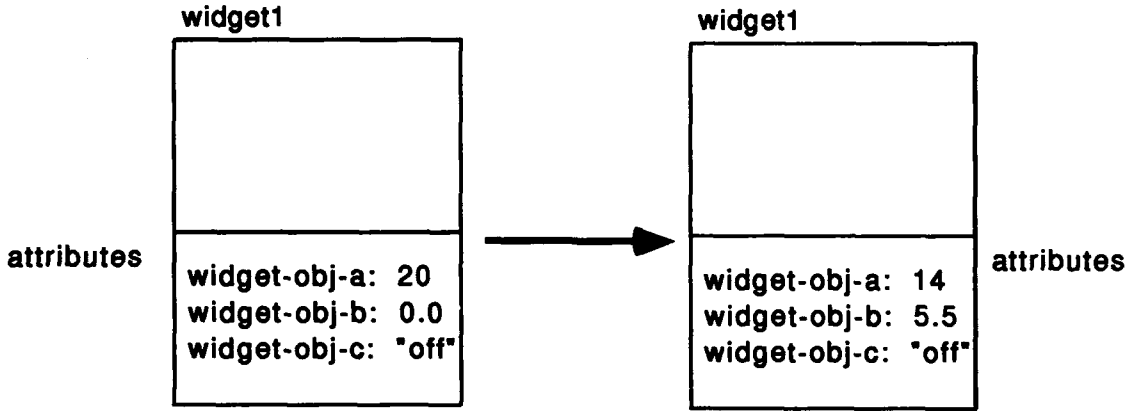

Figure 5.9. SetState Execution

5.3.2 **If** *Statements* **If** statements are executed via **DO-IF-STMT**. If the IF-COND evaluates to true, the statements following the **then** are executed. If the IF-COND evaluates to false, the statements following the **else** are executed (or the statements following the **end if** if no **else** is specified). The condition is evaluated via EVALUATE-BOOLEAN-EXPRESSION; the **then** and **else** statements are executed by a sequence of calls to EXECUTE-STATEMENT.

5.3.3 **While** *Statements* **While** statements are executed via DO-WHILE-STMT. The WHILE-COND is evaluated; if it is true, the statements within the **while** loop are executed via calls to EXECUTE-STATEMENT and then the WHILE-COND is reevaluated. Execution continues in this manner until the WHILE-COND evaluates to false, at which time execution proceeds to the statement following the **end while**.

5.4 Summary

This chapter presented a detailed design of the Preprocess, Semantic Checks, and Execute portions of Architect, the application composition system which was implemented during this research effort. Where relevant, detailed discussions of design considerations and implementation alternatives were presented to explain the decisions that were made.

VI. Validation Domain

To demonstrate the suitability and effectiveness of Architect, the application composition system described in the previous chapters, one must select a domain, conduct a analysis of that domain, construct an appropriate technology for it, and compose useful applications within that domain. This chapter examines the domain that was selected for this validation process: logic circuits. After a discussion of the domain analysis, the validation results are summarized and an assessment of the application composer's performance is presented.

6.1 Background

To further understand the OCU software architecture model and its implications, Architect was first tested using a pedagogical domain consisting of gadgets, widgets, things, contraptions and glibsnitzes. This nonsensical domain enabled us to concentrate on the fundamentals of implementing the OCU model, free of the built-in biases, constraints, and limitations inherent in a "real" domain. This freedom allowed experimentation with various implementation strategies, with the goal of developing a very general approach which could be used successfully on all future application domains.

Because there were no constraining associations between the domain's objects and "real world" entities, domain modeling was trivial. However, an effort was made to provide each class of primitive object with at least one attribute/state data, covering the gamut of REFINE data types to ensure that Architect could effectively handle each one. In addition, object update functions were developed to be fully capable of exercising all aspects of the OCU model.

The knowledge and experience gained through experimentation with this pedagogical domain allowed a smooth transition into the official validation domain.

6.2 Logic Circuit Domain

A subset of the logic circuit domain was chosen as the validating domain for this application composer. It is well-known, well-understood, can be used to compose a wide

variety of practical applications, and the behavior of its components can be easily described (an important consideration given the limited time resources available for this research effort). Refer to Appendix A for a standard template for describing a primitive object within this application generation system. Appendix E contains the logic circuit domain modeled in the REFINE language.

6.2.1 Domain Analysis - Part I

6.2.1.1 Identification of Primitive Objects The first step of the domain analysis was to determine which objects should be included in the validating domain. The following were obvious choices for primitive objects within the domain:

- AND gate
- OR gate
- NAND gate
- NOR gate
- NOT gate

In the real world, none of the above objects has persistent state data that helps to determine the result of its next update; state data is a key aspect of the OCU model from which we developed this application composition system. Although the following component could be constructed from the gates identified above, it was included as a primitive object to provide an example of state data manipulation.

- JK FLIP-FLOP

The implemented application composer uses a simplified application executive which does not support external I/O. Therefore, all data used in the application must be generated within the application and any data produced by the application which is of interest to the application specialist must be handled within the application itself. To accommodate these temporary restrictions, the following objects were included in the domain to generate data and display it to the application specialist, respectively:

- SWITCH
- LED (light emitting diode)

6.2.1.2 Identification of State, Attribute and Constant Data Unlike the OCU model upon which it is based, this implementation makes no distinction among these categories of information which pertain to primitive domain objects. All of these data are treated in the same manner and stored as REFINE object attributes.

- AND gate, OR gate, NAND gate, NOR gate, NOT gate
  - gate delay: integer
  - manufacturer: string
  - meets military specifications?: boolean
  - power required by/consumed by gate: real
- JK FLIP-FLOP
  - gate delay: integer
  - manufacturer: string
  - meets military specifications?: boolean
  - power required by/consumed by gate: real
  - set-up delay: integer
  - hold delay: integer
  - state: boolean
- LED
  - manufacturer: string
  - color of display: symbol
- SWITCH

- manufacturer: string
- debounced?: boolean
- gate delay: integer
- position of switch: symbol (on or off)

6.2.1.3 Identification of Object Update Functions Under the OCU model, an object's behavior is encapsulated in its update function.

- AND gate: The gate's output is the result of the boolean AND operation on its two inputs.
- OR gate: The gate's output is the result of the boolean OR operation on its two inputs.
- NAND gate: The gate's output is the inverse of the boolean AND operation on its two inputs. See Table 6.1.

Table 6.1. Truth Table – NAND gate

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

- NOR gate: The gate's output is the inverse of the boolean OR operation on its two inputs. See Table 6.2.

Table 6.2. Truth Table – NOR gate

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

- NOT gate: The gate's output is the inverse of its input.

- JK FLIP-FLOP: If there is no clock input (i.e., it is false), the flipflop's state does not change. If there is a clock input, the flip-flop's state may change depending on the inputs and its current state. The truth table for a JK FLIP-FLOP appears in Table 6.3.

Table 6.3. Truth Table – JK FLIP-FLOP

clock	J	K	New Q	New $\overline{Q}$
0	x	x	old Q	$\sim (oldQ)$
1	0	0	old Q	$\sim (oldQ))$
1	0	1	0	1
1	1	0	1	0
1	1	1	$\sim (oldQ)$	old Q

- SWITCH: If the switch is in the "on" position, its output is true; if the switch is in the "off" position, its output is false.
- LED: The LED displays, in English, the value of its input.

6.2.1.4 Identification of Object Input-Data and Output-Data As previously described, under the OCU model, input-data represents external data required by an object to effect its update properly; output-data represents data from an object's update which must be made available to other objects in the application. With the identification and description of the update functions for each primitive object, this data is now obvious.

- AND gate, OR gate, NAND gate, NOR gate:
  - Input-data: in1, in2 = signal data whose primitive data type is boolean.
  - Output-data: out1 = signal data whose primitive data type is boolean.
- NOT gate:
  - Input-data: in1 = signal data whose primitive data type is boolean.
  - Output-data: out1 = signal data whose primitive data type is boolean.
- JK FLIP-FLOP:
  - Input-data: J, K, clock = signal data whose primitive data type is boolean.

- Output-data: Q, Q-bar = signal data whose primitive data type is boolean.

- SWITCH:

  - Input-data: none.

  - Output-data: out1 = signal data whose primitive data type is boolean.

- LED

  - Input-data: in1 = signal data whose primitive data type is boolean.

  - Output-data: none.

6.2.2 Domain Analysis – Part II The logic circuit domain, as defined in Section 6.2.1, proved to be inadequate to fully demonstrate all aspects of the application composer implemented during this research effort. Specifically, each primitive object's behavior can be fully specified using a single update function; Architect's capability to dynamically change from one update function to another and, thus, to change an object's behavior could not be demonstrated in a meaningful way. In addition, no primitive objects possessed any coefficient data; the ability to change the effects of a particular update function by changing the value of one or more coefficients used in its calculation could not be shown. This shortcoming in the logic circuit domain was overcome by adding a new primitive object (COUNTER) and by adding an additional update function for the LED object.

6.2.2.1 Identification of Primitive Objects The following primitive object was added to those already identified in Section 6.2.1.1:

- COUNTER

6.2.2.2 Identification of State, Attribute and Constant Data No changes were necessary to this descriptive information previously identified in Section 6.2.1.2. For the newly identified primitive object:

- COUNTER

- gate delay: integer
- manufacturer: string
- meets military specifications?: boolean
- power required by/consumed by gate: real
- count (state data): integer

6.2.2.3 Identification of Object Update Functions Except for LED, all the update functions identified in Section 6.2.1.3 are unchanged.

- LED:
  - T-F-UPDATE: If the input is true, display "true" else display "false".
  - ON-OFF-UPDATE: If the input is true, display "on" else display "off".
- COUNTER: If the reset input is true, set counter to 0 else if the clock input is true, add one to the counter. If the count is greater than the maximum value for the counter, reset counter to 0. The maximum value for the counter is a coefficient; it can be dynamically modified during behavior simulation.

6.2.2.4 Identification of Object Input-Data and Output-Data All input-data and output-data identified in Section 6.2.1.4 remain unchanged. Input-data and output-data for the new primitive object follow:

- COUNTER:
  - Input-data: clock, reset = signal data whose primitive data type is boolean.
  - Output-data: lsb (least significant bit), msb (most significant bit) = signal data whose primitive data type is boolean.

6.2.2.5 Identification of Coefficient Data Coefficients, if applicable for an object, are used in calculating its new state; changing a coefficient can alter the object's behavior or state calculation. The following coefficients apply to this domain:

- COUNTER
  - max-count: Represents the maximum value to which the counter can count. Because the counter's output is limited to two bits (to simplify the connection process), permissible values for max-count are the integers, 0-3. The default value is 3.

6.2.3 Domain Analysis – Part III The logic circuit domain identified and described in Sections 6.2.1 and 6.2.2 is adequate to construct any desired electronic circuit; its primitive objects are the fundamental building blocks of all real-world circuits. However, composing large scale circuits from these very primitive components is tedious, at least partially due to the current lack of an effective visual interface. Inclusion of "higher-level primitives," objects which can be constructed from combinations of existing primitive objects but are treated as primitive objects within the framework of this system, can simplify this tedious connection process for larger circuits. Furthermore, these higher-level primitives illustrate an important concept: what constitutes a "primitive object" depends on the context in which it is to be used.

6.2.3.1 Identification of New Primitive Objects The following "higher-level" primitive objects were added to the logic circuit domain to simplify the connection process for large circuits as well as to illustrate the feasibility/utility of including "higher-level primitives" within a domain:

- DECODER (3-to-8 Line)
- HALF ADDER
- MULTIPLEXER (4-Input MUX)

6.2.3.2 Identification of State, Attribute and Constant Data

- DECODER
  - delay: integer
  - manufacturer: string

- meets military specifications?: boolean
- power required by/consumed by component: real

- HALF ADDER
  - delay: integer
  - manufacturer: string
  - meets military specifications?: boolean
  - power required by/consumed by component: real

- MULTIPLEXER
  - delay: integer
  - manufacturer: string
  - meets military specifications?: boolean
  - power required by/consumed by component: real

6.2.3.3 Identification of New Object Update Functions

- DECODER: The three inputs, taken together, represent a three-digit binary number. One of the eight output lines (numbered 0-7) is set to true depending on the value of this binary number. The truth table for a 3-to-8 line decoder appears in Table 6.4.

Table 6.4. Truth Table – 3-to-8 Line Decoder

X	Y	Z	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

- HALF ADDER: The two inputs, each representing a single binary digit, are added together, producing the sum output. The second output, carry, represents the carry-out and is set if the sum can not be represented in one binary digit. The truth table for a half adder appears in Table 6.5.

Table 6.5. Truth Table – Half Adder

X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- MULTIPLEXER: Two of the inputs (the "select" lines) determine which one of the other four inputs will be used to set the output. The function table for a 4-input multiplexer appears in Table 6.6.

Table 6.6. Truth Table – 4-Input Multiplexer

s_1	s_0	Output
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

6.2.3.4 Identification of New Object Input-Data and Output-Data

- DECODER:
  - Input-data: in1, in2, in3 = signal data whose primitive data type is boolean.
  - Output-data: m0, m1, m2, m3, m4, m5, m6, m7 = signal data whose primitive data type is boolean.
- HALF ADDER:
  - Input-data: in1, in2 = signal data whose primitive data type is boolean.
  - Output-data: s (sum), c (carry) = signal data whose primitive data type is boolean.

- MULTIPLEXER:
  - Input-data: in0, in1, in2, in3, s0, s1 = signal data whose primitive data type is boolean.
  - Output-data: out1 = signal data whose primitive data type is boolean.

6.3 Summary of Results for the Logic Domain

Using Architect, several electronic circuits were constructed from the primitive objects of the logic circuit domain and their behaviors simulated. In all cases, when the application was specified and composed properly (that is, all import-export connections were correctly made), the expected results were achieved. Table 6.7 lists some of the circuits tested during this validation phase and summarizes certain statistics about their compositions. Examples of these composed circuits are contained in Appendix C.

Table 6.7. Summary of Validation Results

Circuit	Number of Primitives	Number of Connections
Decoder from low-level primitives	30	43
Decoder from high-level primitive	12	11
Full Adder	13	16
BCD Adder	43	61
Binary Array Multiplier	14	16
Universal Shift Register	25	44

6.4 Conclusions

The pedagogical domain of widgets, gadgets, etc. proved extremely useful during initial testing of the system's application composer. Its non-association with "real world" entities provided a freedom to fully explore the mechanics of implementing various aspects of the OCU (primarily import/export issues), as well as a suitable base from which to test the system's manipulation of all REFINE primitive data types. A pattern evolved in creating primitive object descriptions which became the standard template to be followed for all such descriptions, regardless of domain. Its nonsensical nature further underscored the need to keep the application composer free of domain-specific references.

The application composer performed very well when the the pedagogical domain was replaced by the logic circuit domain. The semantic analysis and behavior simulation processes required no modifications. However, with the advent of a real domain, shortcomings were identified with the import/export process. Originally, imports were associated with their corresponding exports by matching data names only. In a pedagogical domain that could be constructed in any manner to suit the circumstances, this was no disadvantage. With a real domain, this scheme for matching imports to exports was quickly exposed as inadequate and overly restrictive. Import/export names were retained for internal reference only and an import/export data category was added. The data category, described in Section 5.1.3, serves as the discriminator for determining potential import/export connections; this is its sole function. This system modification, though made in response to the logic circuit domain, is clearly an improvement which will likely satisfy most other application domains.

Proper analysis of any potential domain is essential to maintaining the application composer's domain independence. The composer must be free of inappropriate, domain-specific adaptations which would further complicate software maintenance and limit Architect's usefulness and flexibility. Difficulties in modeling the domain must be carefully evaluated before modifications to the composer are considered to accommodate pecularities of the domain: can the domain be described in different manner to overcome this apparent problem? is the proposed modification applicable to other domains? During logic circuit domain analysis, for example, various aspects of the domain appeared to be difficult to model. In most cases, an alternative modeling approach was found to represent the information within the context of the existing composition system. In one case mentioned in the previous paragraph, the proposed modification was deemed to be appropriate for most other domains as well; the change was incorporated into Architect.

Architect is a domain-independent system. Its straightforward design easily accommodated its extension into the logic circuit domain. Rigorous adherence to the paradigm established by the primitive object template discussed in Appendix A results in primitive objects which fit properly into Architect's framework. This is a flexible system which appears to be capable of being used for a wide variety of application domains.

VII. Conclusions and Recommendations

This chapter provides a summary of the accomplishments of this thesis effort. It also discusses the conclusions which can be drawn from this work and presents some recommendations for further research.

7.1 Summary of Accomplishments

The objective of this research was stated in Chapter I:

> ***Develop a formalized model of a software architecture and implement it within a domain-specific application composition system.***

To that end, the current literature on software architectures was examined and various architectures evaluated in an attempt to find an existing one suitable for composing applications within our system. One such architecture, the Object-Connection-Update model which was developed by the Software Engineering Institute, was studied at length and was ultimately selected:

- It had been used successfully to design and implement various other projects at the SEI and AFIT (23, 7, 40)
- It was described, in considerable detail, in several publications (23, 24, 42)
- It was capable of supporting our application composition system.

The OCU model was formalized using the REFINE wide-spectrum language into a formally specified, executable prototype. This prototype of the application composer was validated using the logic circuit domain.

7.2 Conclusions

The following general conclusions can be drawn from this research:

1. Application composition systems, such as that described in Chapter III, are feasible. This was clearly demonstrated in Appendix C, where several complex logic circuits were constructed using Architect.

2. Non-programmers, who are very knowledgeable about a particular domain, *can* create quite sophisticated applications without the direct assistance of software professionals. These application specialists, armed only with detailed knowledge about their own domains and an application composition system with an appropriately modeled domain-specific technology base, can quickly construct effective (they behave as intended) applications to satisfy virtually any requirement within the domain.

3. A suitable software architecture is an essential ingredient of a flexible, domain-independent application composition system. The software architecture allows the system/application designer to concentrate on the fundamental elements of constructing a system/application: *what* components must be included and *what* connections are appropriate among those components. *How* those connections are actually made is the software architecture's concern. This separation of concerns allows the focus to be on *what* should be constructed, not *how* it is to be implemented; we can expect such a focus to produce better designed, more reliable systems.

The following specific conclusions can be drawn about Architect, the application composition system which was developed during this thesis effort:

1. Architect works. A wide range of electronic circuits were constructed during testing, some of which are presented in Appendix C.

2. Architect is readily extensible. Initially, to demonstrate the feasibility of the system concept and to evaluate various implementation strategies (primarily for import/export areas), a pedigogical domain was used. It was a simple matter to reorient the system to the logic circuit domain, an effort which required only three manhours (the time needed to created a new domain-specific language and technology base). Changing domains required no modifications to the composition system source code; only the domain-specific technology base (and DSL) required changes.

The template in Appendix A should be helpful to software engineers who must maintain the technology base.

3. Domain analysis is critical. An application composition system provides only the framework around which domain applications can be constructed; the technology base supplies the details. The contents of the technology base depend directly upon the results of a domain analysis. An ideal domain analysis will identify all the appropriate objects within the domain and will describe them properly (attributes, input-data, output-data, update functions, etc.); virtually any meaningful application within the domain can be constructed when all the necessary objects are properly identified. A typical domain analysis will omit some necessary domain objects and/or improperly describe them; applications can not be composed correctly if the necessary components are not available and can not behave as required if the components are incorrectly defined. The good news: because Architect is easily extensible, oversights and incorrectly defined objects can be quickly fixed once identified.

4. Software engineers who work with Architect must clearly understand the OCU model and its implementation. A domain analysis may identify a situation which apparently can not be accommodated by the OCU model. For example, if a primitive object has two update functions, each of which performs a different operation with different input-data and output-data (a situation which can not be handled by the model), an unwary software engineer might be tempted to change the implementation to accommodate it.

5. Making connections between import items and the export items that are to provide their data requests can be very tedious in some domains using the current system; a glance at some of the application specifications in Appendix C clearly illustrates this point. The visual system will alleviate this problem. However, even without the visual system, meaningful, complex applications can be constructed; it just takes a bit of time and effort.

6. The use of the Software Refinery environment, with its integrated language definition facility (DIALECT), graphical user interface (INTERVISTA), and programming language (REFINE) which incorporates many built-in object manipulation functions,

greatly simplified and expedited the implementation of Architect. With DIALECT, parsers for domain-specific and architecture languages were easily generated from simple BNF descriptions of the languages. Output from these DIALECT-generated parsers was automatically converted into an abstract syntax tree format and stored in the structured object base. The REFINE language's built-in functions provide an easy way to manipulate the object base and its standard programming language constructs supply the functionality expected of any high-level programming language; in fact, its direct support of set theory and set-based operations provides more power than can be achieved with most commonly used languages. The availability of these powerful, well-integrated tools eliminated the need to write comparable tools; this significantly shortened the development process.

7.3 Recommendations for Further Research

The following issues should be addressed in future reseach efforts:

1. Code generation – The application composition system described in Chapter III includes a formal specification generation capability which is intended to feed an automatic code generator. Currently, the behaviors of application specifications, which are created by application specialists, can be simulated to verify that behavior. However, this is merely a simulation and is neither robust nor efficient enough to support a production system.

2. Extending Domains – Architect, as currently implemented, does not provide any automated support for extending the domain knowledge which resides in the domain model. Extensions to the domain model must be made manually and are limited by the knowledge and understanding of the domain engineer. The Kestrel Interactive Development System (KIDS) should be studied to ascertain its ability to allow automated extensions to Architect's domain models.

3. Application executive – Only a very simple application executive was considered in this implementation. This "application executive" was merely a specialized, highest level subsystem. Obviously, such a simple approach is inadequate for most mean-

ingful, production domains. What should be the role of an executive? How does the executive interface with the environment to obtain the necessary external data? Can the executive perform in a real-time environment with concurrency and time constraints? These questions must be answered before a full-scale, production application composition system can be developed.

4. Additional validating domains – We have demonstrated that an application composition system is feasible. However, application compositions need to be attempted in a wider variety of real-world domains to further assess its strengths and shortcomings. One domain to consider for such further evaluation is the simulation domain for the Joint Modeling and Simulation System (J-MASS).

5. Alternatives to the OCU model – The complete OCU model as described in (24) has not been implemented. Certain changes (for example, the omission of several subsystem and object procedural interfaces) were made to the model to accommodate the time limitation of this research effort and to conform with certain predetermined requirements (e.g., that all objects would be created during the specification, not dynamically at run-time). Other changes should also be considered. For example,

   - the model does not currently allow a subsystem to directly query a subordinate object concerning its state data; this would seem to be a desirable feature, especially when dealing with `if` and `while` statement conditions.
   - there is currently no way to "hide" exports inside a subsystem (that is, to prevent them from being used outside the subsystem; all exports are treated alike and may be used for the source of *any* compatible import. Within the validating domain, there were many instances where "intermediate" outputs were produced and consumed within a single subsystem; there was never any intention to use these "intermediate" results in any other subsystem. In fact, using such "intermediate" results would likely produce an incorrect composition. But, in keeping to the OCU model, all exports are globally accessable.

   The OCU model must be carefully studied within the context of this application composition system and its suitability further evaluated. Additional tests may dis-

cover that it is too restrictive (or permissive) to allow appropriate compositions for all domains of interest.

7.4 Final Comments

The application composition system developed during this thesis effort is a significant first step in a software development revolution. Software engineers will no longer develop systems to satisfy a single, unique requirement (complicated though it may be); end users will create their own applications, without intervention by computer professionals. Long waits for inadequate, often unreliable and incorrect software products will be only a distant memory. Software maintenance, once the major expense in any software system's lifecycle, will be an issue no longer; application specifications, not source code, will be maintained by the end users themselves. Software development teams will be composed of domain engineers, software engineers, and application specialists; knowledge about domains must be formalized, application composition and generation systems must be developed/maintained, and problems within a domain must be analyzed, evaluated and solved.

Appendix A. *Requirements for Specifying Primitive Objects*

Architect, which was implemented in this research effort, is predicated on the assumption that all primitive objects are defined in a precise, standardized manner by the software engineer. This appendix provides a template to be used by the software engineer when creating a definition for all primitive object classes within any domain and explains the significance of the mandatory items.

INPUT- DATA	{(name, category, type-data, ,)... }
OUTPUT- DATA	{(name, category, type-data, ,)... }
COEFFICIENTS	{(name, value) ... }
UPDATE- FUNCTION	function-name
Attributes, Current-State, Constants	
$variable_1$ $variable_2$ . . $variable_n$	

Figure A.1. Standard Primitive Object Definition

A.1 Primitive Object Definition Template

Figure A.1 illustrates the standard template for all primitive object definitions. It is based on the kinds of data available to primitive objects as described by the OCU model.

See Appendix E for examples of primitive object definitions from the logic circuit domain used to validate this implementation.

A.1.1 INPUT-DATA INPUT-DATA describes the data which is external to the object but is needed to update it. INPUT-DATA is implemented as a set of IMPORT-OBJ objects; during preprocessing (BUILD-IMPORT-EXPORT-AREA), each entry in the set becomes a part of the import area of the subsystem which "controls" that primitive object.

- IMPORT-NAME: identifies the name by which this piece of data will be referred. In the object's update function, this name must appear in a GET-IMPORT function call to obtain the data's actual value. Additionally, the name will be displayed to the application specialist during preprocessing (BUILD-IMPORT-SOURCES) to uniquely identify this input-data item when more than one piece of external data can serve as its source.
- IMPORT-CATEGORY: identifies the type of the external data required, in domain-oriented terms. For example, one might specify that the data must be of the category "temperature" or "time," rather than merely a real number or an integer. This is analogous to the Ada programming language which encourages the use of subtypes to further constrain the possible values which a given variable can accept. Only EXPORT-OBJs with the same category can be considered as potential sources for this input-data item.
- IMPORT-TYPE-DATA: identifies the primitive data type of the required data. This data type is used only for checking and evaluating the expressions in `if` and `while` statements. The current implementation accommodates only primitive data types (integer, real, boolean, string and symbol).

A.1.2 OUTPUT-DATA OUTPUT-DATA describes the data which the object must make available externally to other application components. It is implemented as a set of EXPORT-OBJ objects; during preprocessing (BUILD-IMPORT-EXPORT-AREA), each

entry in the set becomes a part of the export area of the subsystem which "controls" that primitive object.

- EXPORT-NAME: identifies the name by which this piece of data will be referred. In the object's update function, this name must appear in a SET-EXPORT function call to make the new value accessible to other application components. Additionally, the name will be displayed to the application specialist during preprocessing (BUILD-IMPORT-SOURCES) to uniquely identify this output-data as a possible source when more than EXPORT-OBJ can serve as the source for an IMPORT-OBJ.
- EXPORT-CATEGORY: identifies the type of the external data (OUTPUT-DATA) produced, in domain-oriented terms. OUTPUT-DATA can be used only by those import items which have the same category.
- EXPORT-TYPE-DATA: identifies the primitive data type of the required data. This data type is used only for checking and evaluating the expressions in `if` and `while` statements. The current implementation accommodates only primitive data types (integer, real, boolean, string and symbol).

A.1.3 COEFFICIENTS In the OCU model, coefficients represent data which can be used in an object's update function to alter the behavior or performance of the object. In this implementation, coefficients are expected to have a default value, determined by the domain analysis, and can be modified, as necessary, at any point in the execution via a `setfunction` statement in the subsystem's update procedure.

A coefficient is represented as a NAME-VALUE-OBJ: NAME-VALUE-NAME is the name of the coefficient (to be used in the update function when referencing this coefficient) and NAME-VALUE-VALUE is the current value associated with the coefficient. The coefficient's value is not constrained by a particular data type; rather, it is implemented as a REFINE ANY-TYPE which, as its name implies, allows any type of data to be stored. This requires that the software engineer ensure compatibility of data types between default coefficient values and their usage in object update functions. In addition, it imposes a responsibility on the application specialist to provide compatible data when specifying

new values for coefficients in **setfunction** statements; no semantic checks can ensure data consistency before behavior simulation. This may appear to unduly burden the application specialist; however, this approach provides the flexibility needed to accommodate any potential domain's requirements for number and type of coefficients.

A.1.4 UPDATE-FUNCTION This variable stores the name of the function currently used to update the primitive object. During behavior simulation, when an **update** statement for a primitive object is encountered, Architect retrieves the name of the function to be used for updating from this variable and calls the indicated function to complete the operation. It is expected that domain analysis will have identified a "normal" or default update function for each primitive object class. An alternate update function can be specified at any time during behavior simulation via a **setfunction** statement in the subsystem's update procedure; subsequent **update** statements applied to that object will use this new update function.

A.1.5 Attributes, Current_State, Constants Although the OCU model considers these to be different kinds of data, this implementation makes no such distinctions; all are modeled as REFINE attributes of a primitive object. A strict interpretation of the OCU model would allow only current_state data to be modified directly by a **setstate** statement; the current implementation allows any of this data to be changed. If it becomes necessary or desirable to do so, enforcing these distinctions could be accomplished via a naming convention scheme; as an example, attributes (object characteristics) could be represented with the letters "ATTR" imbedded within attribute names, "STATE" within current_state names and "CONST" within the names of constants.

A.1.6 Miscellaneous

- All primitive object definitions must begin with an object class specification. This implementation assumes that each primitive object class name consists of the kind of real-world object represented by the class followed by "-OBJ" (e.g. AND-GATE-OBJ). This class name (minus the "-OBJ") appears in the domain-specific grammar

when specifying (i.e., creating) the object instances which are to be part of the application.

- All variable names associated with an object are prefaced by the complete object class name. This ensures that variable names are unique – the software engineer can use descriptive, domain-oriented names for all variables without being concerned that some other class of primitive object might already have a variable with the same name. It also allows the application specialist to refer to variables in **setstate** statements by just this domain-oriented name; Architect can assemble the entire variable name by prefacing the given name with the object class of the statement's operand.
- All update functions are included in the primitive object definition. Each update function is coded as a REFINE function whose parameters include the subsystem which controls the primitive object being updated and the primitive object itself.

Appendix B. *Guide to Using the Application Composer*

Maintaining a computer program, especially one written by a colleague who is no longer available for consultation, can be a daunting task. This appendix attempts to ease that burden somewhat by providing detailed, technical information needed to execute the application composer. The intended users of this guide are the software engineers who will be tasked to extend the capabilities of this composer. It is not written for the software engineers who create and maintain domain-specific languages and technology bases; helpful information of this nature can be found in Appendix B of (33).

B.1 Getting Started

The application composer must execute within the Software Refinery environment which must be accessed through an Emacs process. To enter the Software Refinery Environment:

1. From a command or shell window, set the current directory to that which contains Architect.
2. Invoke Emacs (**emacs** or **emacs&** to run in the background).

After the large Emacs window appears, start Software Refinery by pressing **<esc>** (which causes the cursor to jump to the lower left corner of the Emacs window) and typing **run-refine**. After a short initialize phase during which various messages will be displayed on the right side of the Emacs window, Software Refinery will be ready for use.

The application composer's executable code modules (suffixed by **.fasl** must be loaded before Architect can be executed; if no executable modules exist, they must be created via a compile step. There are many dependencies among the many files which comprise Architect. Therefore, to ensure all files are compiled and/or loaded in the proper order, it is recommended that a "load" file (to load all executable modules) and a "compile-and-load" file (to compile source code modules and load the newly created executable modules) be used. The "compile- and-load" file for Architect is listed in Section B.3.

A "compile-and-load" file is actually a lisp function whose name follows **defun** (this discussion also applies to "load" files). This function must be loaded into the Refine object base before it can be executed. This is accomplished by typing `(load "cl")` at the Refine prompt (`.>`), where `cl` corresponds to the name of the lisp function. After the function is loaded, it can be executed by typing `(cl)`. Execution of the function causes each designated file to be compiled and loaded, in turn. Note: the DIALECT system must be loaded first (`(load-system "dialect" "1-0")`).

B.2 Using the Application Composition System

Now that Architect is loaded, it is ready for use. If the user wants to employ the full capabilities of the composition system, he should refer to the instructions in Appendix A of (33).

However, if the user's focus is strictly the application composer itself (i.e., he is not interested in using generic components, loading previously saved components/architectural fragments, nor editing application definitions), the easiest and fastest method of populating the structured object base is via parsing. This is accomplished using the left side of the Emacs window. First, establish it as the "active" window by moving the cursor there and clicking the right mouse button. Then type as desired using the Emacs editor. Or, textual application definitions may be loaded from any file by typing `<ctrl> x <ctrl> f`, and completing the file pathname that appears at the bottom of the window. When the desired application definition appears in the left window, parse it into the Refine object base as follows:

1. Move the cursor to the beginning of the definition and type `<esc><space>` to mark the beginning of the parse block.
2. Move the cursor just beyond the end of the definition and type `esc><space>` to mark the end of the block.
3. Move the cursor back to the top of the block and `<esc> w` to move the block into the Emacs buffer.
4. Move the cursor to the right side of the Emacs window, establish it as the "active" window, and type at the Refine prompt:

```
(\#> <ctrl>y <ctrl> )
```

Note: `<ctrl> y` causes the contents of the Emacs buffer to be "dumped" into a the Refine parse command (`#>`).

Upon successful completion of the parse command, the application definition is stored as an abstract syntax tree (AST) in the Refine object base. The root of that AST is now the "current node;" this is significant as Refine `rules` operate only on the current node. In the present implementation, the user interacts with Architect by invoking applicable rules. The list of currently applicable rules can be obtained by typing `(rs)` ("rule search"). After deciding what he wants Architect to do, the user applies the chosen rule by typing `(ar n)` where `n` is the number of the appropriate rule. The user is prompted through any interactions which result from applying that rule.

If the user is interested only in the preprocess, semantic-check, and execute system components, a simplified the application composer can be used. The required models for the simplified version are listed in their compilation order in Figure B.1. It should be noted that the user never loses the ability to "edit," "load," and "store" application definitions, even with this simplified system. Because *all* objects must have unique names, reparsing a modified copy of the original application has the effect of "editing" the application. We have already seen how application definitions can be "loaded" from a text file; application definitions can be "saved" into a text file via the Emacs command, `<ctrl> x <ctrl> s`.

B.3 "Compile-And-Load" File for the Application Composition System

Including the following "compile-and-load" file into this user's guide accomplishes two objectives:

1. It lists the files which are required to execute Architect within the logic circuit domain.
2. It establishes a compilation order to accommodate program dependencies.

```
\begin{singlespace}
(defun cl()
```

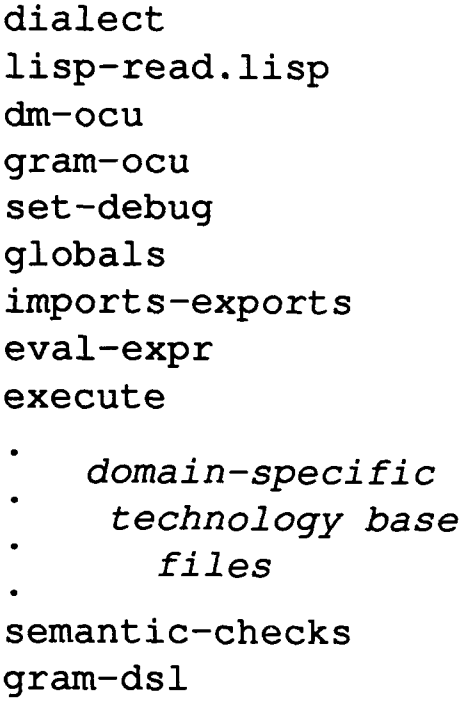

Figure B.1. Compilation Order for Simplified Application Composer System

```
(load-system "dialect" "1-0")

(compile-file "./OCU-dm/dm-ocu")
(load "./OCU-dm/dm-ocu")

(compile-file "./OCU-dm/gram-ocu")
(load "./OCU-dm/gram-ocu")

(compile-file "./DSL/globals")
(load "./DSL/globals")

(compile-file "./DSL/lisp-utilities.lisp")
(load "./DSL/lisp-utilities.lisp")

(compile-file "./DSL/obj-utilities")
(load "./DSL/obj-utilities")

(compile-file "./DSL/read-utilities")
(load "./DSL/read-utilities")

(compile-file "./DSL/erase")
(load "./DSL/erase")

(compile-file "./DSL/menu")
(load "./DSL/menu")

(compile-file "./DSL/display-files")
(load "./DSL/display-files")

(compile-file "./DSL/modify-obj")
```

```
(load "./DSL/modify-obj")

(compile-file "./DSL/save")
(load "./DSL/save")

(compile-file "./DSL/generic")
(load "./DSL/generic")

(compile-file "./DSL/build-generic")
(load "./DSL/build-generic")

(compile-file "./DSL/complete")
(load "./DSL/complete")


(compile-file "./OCU/set-debug")
(load "./OCU/set-debug")

(compile-file "./OCU/imports-exports")
(load "./OCU/imports-exports")

(compile-file "./OCU/eval-expr")
(load "./OCU/eval-expr")

(compile-file "./OCU/execute")
(load "./OCU/execute")

(compile-file "./OCU/semantic-checks")
(load "./OCU/semantic-checks")
(compile-file "./domain-model/and-gate")
```

```
(load "./domain-model/and-gate")

(compile-file "./domain-model/or-gate")
(load "./domain-model/or-gate")

(compile-file "./domain-model/nand-gate")
(load "./domain-model/nand-gate")

(compile-file "./domain-model/nor-gate")
(load "./domain-model/nor-gate")

(compile-file "./domain-model/not-gate")
(load "./domain-model/not-gate")

(compile-file "./domain-model/switch")
(load "./domain-model/switch")

(compile-file "./domain-model/jk-flip-flop")
(load "./domain-model/jk-flip-flop")

(compile-file "./domain-model/led")
(load "./domain-model/led")

(compile-file "./domain-model/counter")
(load "./domain-model/counter")

(compile-file "./domain-model/decoder")
(load "./domain-model/decoder")

(compile-file "./domain-model/half-adder")
```

```
  (load "./domain-model/half-adder")

  (compile-file "./domain-model/mux")
  (load "./domain-model/mux")

  (compile-file "./domain-model/gram-logic")
  (load "./domain-model/gram-logic")


  (in-grammar 'circuits)
 )
\end{singlespace}
```

Appendix C. *Validation Test Cases and Results*

This appendix presents a subset of the circuits from the logic circuit domain which were constructed to demonstrate the utility of Architect, the application generator implemented during this research. Each test case is presented in the following consistent format: the objective to be achieved, an illustration of the circuit/application to be tested, the application specification written in the domain-specific language and system/user dialogues during the test. Please note: the system/user dialogues have been editted. During actual execution of Architect, the complete list of possible import sources is presented to the application specialist each time one is requested; I have retained only the first such display, deleting the repetitive ones to conserve paper.

C.1 Decoder Test

This test case consists of two independent 3-to-8 line decoders: one constructed from very low-level logic gates (AND, NOT), the other from the domain's decoder primitive object. Each decoder was provided the same input values to demonstrate the equivalence of the two circuits. It also includes a second execution of the same application with different input values. This illustrates two points: 1) the application works correctly with a different set of values and 2) a different user interface is used when import-to-export connections already exist. An additional point should be noted: the primitive decoder is much easier to use than the one constructed from low-level logic gates. This should be a fundamental lesson for the domain engineer: if a particular subsystem will be used repeatedly by application specialists working within the domain, it may be advisable to encapsulate that subsystem into a "high-level primitive" to simplify subsequent application specifications.

C.1.1 Circuit Diagram See Figure C.1 for a decoder implemented as a subsystem composed from low-level logic gates. Figure C.2 illustrates a decoder primitive. Both have been included in the specification for this test case.

C.1.2 Application Specification - Test 1

```
application definition test-decoders-primitive-and-subsystem

switch sub-z  position: on
switch sub-y  position: on
switch sub-x  position: on
switch z      position: on
switch y      position: on
switch x      position: on

not-gate not-sub-z
not-gate not-sub-y
not-gate not-sub-x

and-gate and01
and-gate and02
and-gate and11
and-gate and12
and-gate and21
and-gate and22
```

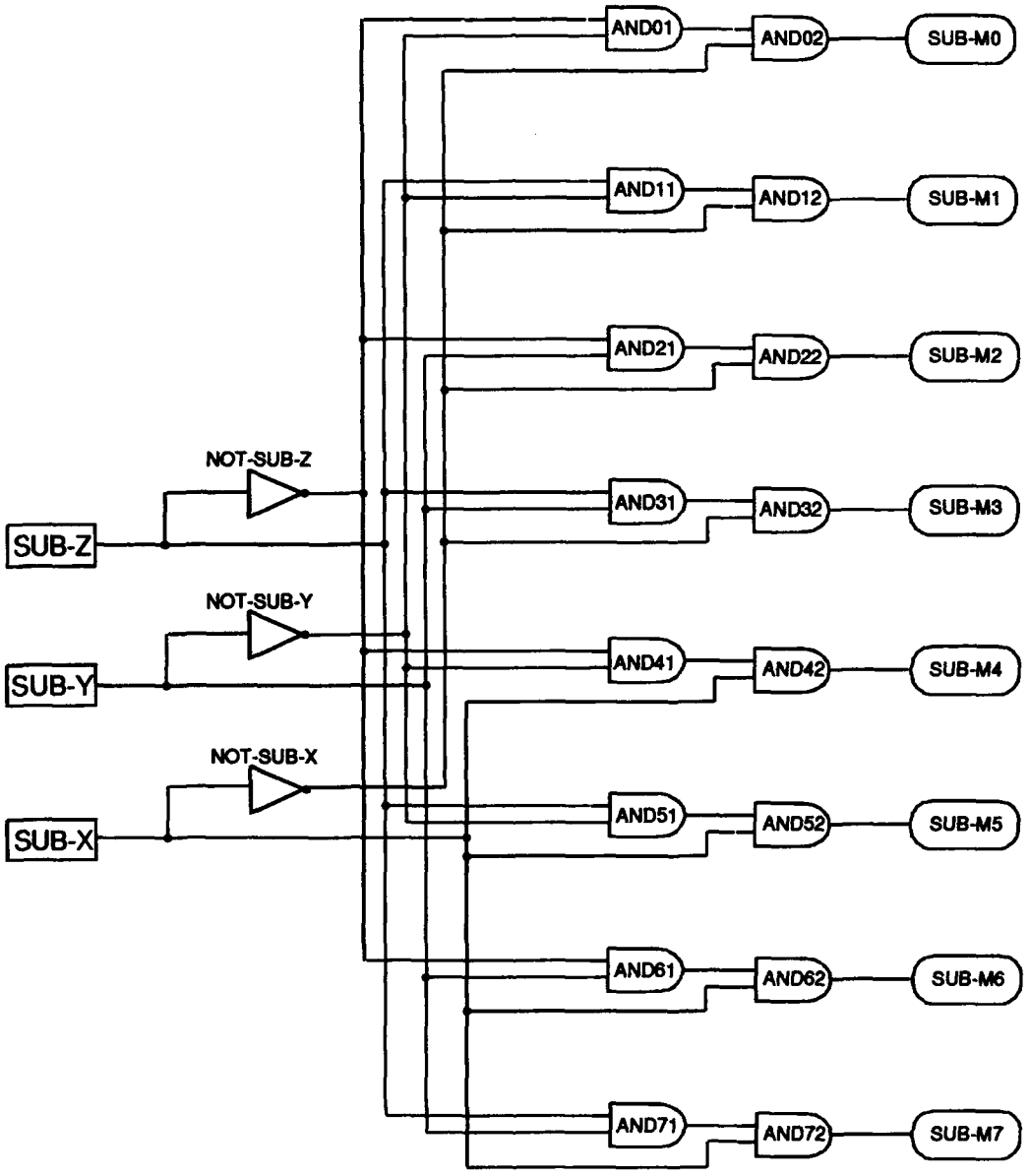

Figure C.1. 3-to-8 Line Decoder (Subsystem)

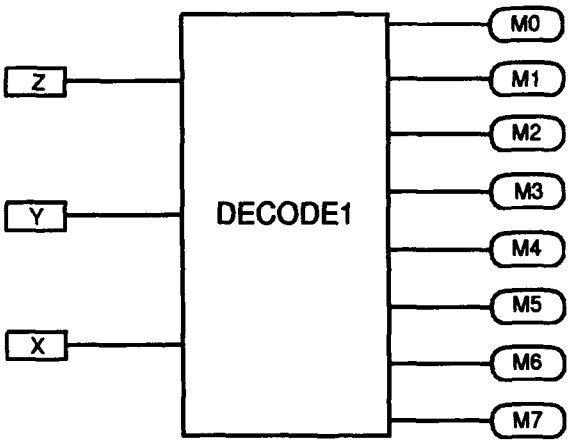

Figure C.2. 3-to-8 Line Decoder (Primitive)

```
and-gate and31
and-gate and32
and-gate and41
and-gate and42
and-gate and51
and-gate and52
and-gate and61
and-gate and62
and-gate and71
and-gate and72

led sub-m0
led sub-m1
led sub-m2
led sub-m3
led sub-m4
led sub-m5
led sub-m6
led sub-m7

led m0
led m1
led m2
led m3
led m4
led m5
led m6
led m7

decoder DECODE1

application decoder-tests is
```

```
   controls:  decoder-subsystem,
              decoder-primitive
   update procedure:
     update decoder-subsystem
     update decoder-primitive

subsystem decoder-subsystem is
   controls:  sub-z, sub-y, sub-x, not-sub-z, not-sub-y, not-sub-x,
              and01, and02, and11, and12, and21, and22, and31, and32,
              and41, and42, and51, and52, and61, and62, and71, and72,
              sub-m0, sub-m1, sub-m2, sub-m3, sub-m4, sub-m5, sub-m6,
              sub-m7
  update procedure:
     update sub-z
     update sub-y
     update sub-x
     update not-sub-z
     update not-sub-y
     update not-sub-x
     update and01
     update and02
     update and11
     update and12
     update and21
     update and22
     update and31
     update and32
     update and41
     update and42
     update and51
     update and52
     update and61
     update and62
     update and71
     update and72
     update sub-m0
     update sub-m1
     update sub-m2
     update sub-m3
     update sub-m4
     update sub-m5
     update sub-m6
     update sub-m7

subsystem decoder-primitive is
    controls: x, y, z, DECODE1, m0, m1, m2, m3, m4, m5, m6, m7
    update procedure:
      update z
      update y
      update x
      update DECODE1
```

```
update m0
update m1
update m2
update m3
update m4
update m5
update m6
update m7
```

C.1.3 System/User Dialogue - Test 1

```
.> (#> application definition test-decoders-primitive-and-subsystem)
application definition
  TEST-DECODERS-PRIMITIVE-AND-SUBSYSTEM
  SUB-Z SUB-Y SUB-X Z Y X NOT-SUB-Z NOT-SUB-Y NOT-SUB-X AND01
    AND02 AND11 AND12 AND21 AND22 AND31 AND32 AND41 AND42 AND51
    AND52 AND61 AND62 AND71 AND72 SUB-M0 SUB-M1 SUB-M2 SUB-M3
    SUB-M4 SUB-M5 SUB-M6 SUB-M7 M0 M1 M2 M3 M4 M5 M6 M7 DECODE1
    DECODER-TESTS DECODER-SUBSYSTEM DECODER-PRIMITIVE
.> (rs)
- Rules for: application definition
  TEST-DECODERS-PRIMITIVE-AND-SUBSYSTEM
  SUB-Z SUB-Y SUB-X Z Y X NOT-SUB-Z NOT-SUB-Y NOT-SUB-X AND01
    AND02 AND11 AND12 AND21 AND22 AND31 AND32 AND41 AND42 AND51
    AND52 AND61 AND62 AND71 AND72 SUB-M0 SUB-M1 SUB-M2 SUB-M3
    SUB-M4 SUB-M5 SUB-M6 SUB-M7 M0 M1 M2 M3 M4 M5 M6 M7 DECODE1
    DECODER-TESTS DECODER-SUBSYSTEM DECODER-PRIMITIVE -
2) CHECK-SEMANTICS
.> (ar 2)

More than one export can provide the data for IN1
                    which is used by object SUB-M7
                                in subsystem DECODER-SUBSYSTEM
Choose the export item (subsystem and component)
  that you wish to be the source of this data:
     1> subsystem "DECODER-SUBSYSTEM" component "SUB-Z" name "OUT1"
     2> subsystem "DECODER-SUBSYSTEM" component "SUB-Y" name "OUT1"
     3> subsystem "DECODER-SUBSYSTEM" component "SUB-X" name "OUT1"
     4> subsystem "DECODER-SUBSYSTEM" component "NOT-SUB-Z" name "OUT1"
     5> subsystem "DECODER-SUBSYSTEM" component "NOT-SUB-Y" name "OUT1"
     6> subsystem "DECODER-SUBSYSTEM" component "NOT-SUB-X" name "OUT1"
     7> subsystem "DECODER-SUBSYSTEM" component "AND01" name "OUT1"
     8> subsystem "DECODER-SUBSYSTEM" component "AND02" name "OUT1"
     9> subsystem "DECODER-SUBSYSTEM" component "AND11" name "OUT1"
     10> subsystem "DECODER-SUBSYSTEM" component "AND12" name "OUT1"
     11> subsystem "DECODER-SUBSYSTEM" component "AND21" name "OUT1"
     12> subsystem "DECODER-SUBSYSTEM" component "AND22" name "OUT1"
     13> subsystem "DECODER-SUBSYSTEM" component "AND31" name "OUT1"
     14> subsystem "DECODER-SUBSYSTEM" component "AND32" name "OUT1"
     15> subsystem "DECODER-SUBSYSTEM" component "AND41" name "OUT1"
     16> subsystem "DECODER-SUBSYSTEM" component "AND42" name "OUT1"
```

```
      17> subsystem "DECODER-SUBSYSTEM" component "AND51" name "OUT1"
      18> subsystem "DECODER-SUBSYSTEM" component "AND52" name "OUT1"
      19> subsystem "DECODER-SUBSYSTEM" component "AND61" name "OUT1"
      20> subsystem "DECODER-SUBSYSTEM" component "AND62" name "OUT1"
      21> subsystem "DECODER-SUBSYSTEM" component "AND71" name "OUT1"
      22> subsystem "DECODER-SUBSYSTEM" component "AND72" name "OUT1"
      23> subsystem "DECODER-PRIMITIVE" component "X" name "OUT1"
      24> subsystem "DECODER-PRIMITIVE" component "Y" name "OUT1"
      25> subsystem "DECODER-PRIMITIVE" component "Z" name "OUT1"
      26> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M0"
      27> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M1"
      28> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M2"
      29> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M3"
      30> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M4"
      31> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M5"
      32> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M6"
      33> subsystem "DECODER-PRIMITIVE" component "DECODE1" name "M7"
      34> Specific source not required; use arbitrary one
Enter the number corresponding to the source you want to use
22

More than one export can provide the data for IN1
                    which is used by object SUB-M6
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
20

More than one export can provide the data for IN1
                    which is used by object SUB-M5
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
18

More than one export can provide the data for IN1
                    which is used by object SUB-M4
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
16

More than one export can provide the data for IN1
                    which is used by object SUB-M3
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
14

More than one export can provide the data for IN1
                    which is used by object SUB-M2
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
12
```

```
More than one export can provide the data for IN1
                    which is used by object SUB-M1
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
10

More than one export can provide the data for IN1
                    which is used by object SUB-M0
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
8

More than one export can provide the data for IN2
                    which is used by object AND72
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                    which is used by object AND72
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
21

More than one export can provide the data for IN2
                    which is used by object AND71
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN1
                    which is used by object AND71
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN2
                    which is used by object AND62
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                    which is used by object AND62
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
19

More than one export can provide the data for IN2
                    which is used by object AND61
                                in subsystem DECODER-SUBSYSTEM
```

```
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN1
                    which is used by object AND61
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN2
                    which is used by object AND52
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                    which is used by object AND52
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
17

More than one export can provide the data for IN2
                    which is used by object AND51
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN1
                    which is used by object AND51
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN2
                    which is used by object AND42
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                    which is used by object AND42
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
15

More than one export can provide the data for IN2
                    which is used by object AND41
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
5
```

```
More than one export can provide the data for IN1
                    which is used by object AND41
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN2
                    which is used by object AND32
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for IN1
                    which is used by object AND32
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
13

More than one export can provide the data for IN2
                    which is used by object AND31
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN1
                    which is used by object AND31
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN2
                    which is used by object AND22
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for IN1
                    which is used by object AND22
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
11

More than one export can provide the data for IN2
                    which is used by object AND21
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN1
                    which is used by object AND21
                                in subsystem DECODER-SUBSYSTEM
```

```
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN2
                    which is used by object AND12
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for IN1
                    which is used by object AND12
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
9

More than one export can provide the data for IN2
                    which is used by object AND11
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN1
                    which is used by object AND11
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN2
                    which is used by object AND02
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for IN1
                    which is used by object AND02
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
7

More than one export can provide the data for IN2
                    which is used by object AND01
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN1
                    which is used by object AND01
                                in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
4
```

```
More than one export can provide the data for IN1
                      which is used by object NOT-SUB-X
                                  in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                      which is used by object NOT-SUB-Y
                                  in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN1
                      which is used by object NOT-SUB-Z
                                  in subsystem DECODER-SUBSYSTEM
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN1
                      which is used by object M7
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
33

More than one export can provide the data for IN1
                      which is used by object M6
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
32

More than one export can provide the data for IN1
                      which is used by object M5
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
31

More than one export can provide the data for IN1
                      which is used by object M4
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
30

More than one export can provide the data for IN1
                      which is used by object M3
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
29

More than one export can provide the data for IN1
                      which is used by object M2
                                  in subsystem DECODER-PRIMITIVE
```

```
Enter the number corresponding to the source you want to use
28

More than one export can provide the data for IN1
                      which is used by object M1
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
27

More than one export can provide the data for IN1
                      which is used by object M0
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
26

More than one export can provide the data for IN3
                      which is used by object DECODE1
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
25

More than one export can provide the data for IN2
                      which is used by object DECODE1
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
24

More than one export can provide the data for IN1
                      which is used by object DECODE1
                                  in subsystem DECODER-PRIMITIVE
Enter the number corresponding to the source you want to use
23
Rule successfully applied.
application definition
  TEST-DECODERS-PRIMITIVE-AND-SUBSYSTEM
  SUB-Z SUB-Y SUB-X Z Y X NOT-SUB-Z NOT-SUB-Y NOT-SUB-X AND01
    AND02 AND11 AND12 AND21 AND22 AND31 AND32 AND41 AND42 AND51
    AND52 AND61 AND62 AND71 AND72 SUB-M0 SUB-M1 SUB-M2 SUB-M3
    SUB-M4 SUB-M5 SUB-M6 SUB-M7 M0 M1 M2 M3 M4 M5 M6 M7 DECODE1
    DECODER-TESTS DECODER-SUBSYSTEM DECODER-PRIMITIVE
```

C.2 Full Adder Test

This case tests a full adder. It is constructed of two half adders: one is a subsystem composed of low-level primitives, the other is the domain's "higher level primitive." This test demonstrates the interchangeability of subsystem versus "higher level primitives" in application specifications.

C.2.1 Circuit Diagram See Figure C.3.

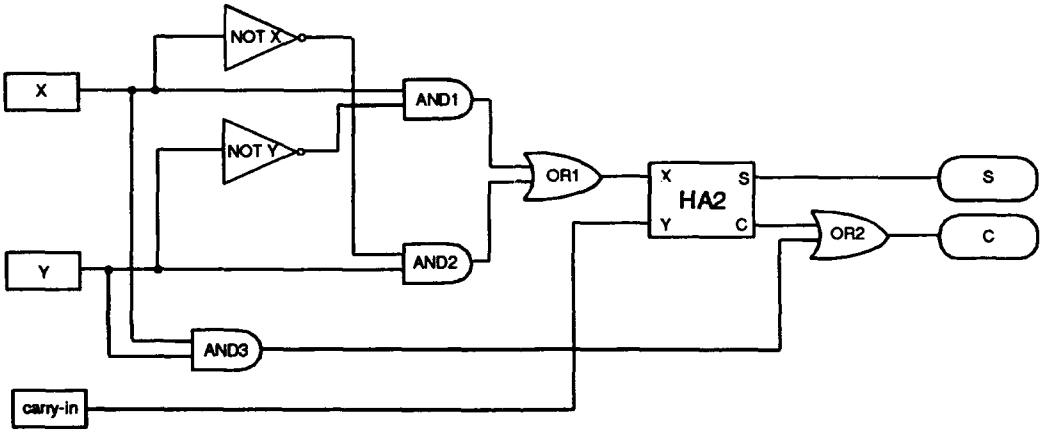

Figure C.3. Full Adder

C.2.2 Application Specification

```
application definition test-full-adder

switch x position: on
switch y position: on
switch carry-in position: off

not-gate not-x
not-gate not-y

and-gate and1
and-gate and2
and-gate and3

or-gate or1
or-gate or2
```

```
led s
led c

half-adder HA2

application full-adder-test is
  controls: full-adder
  update procedure:
      update full-adder

subsystem full-adder is
  controls: half-adder-subsystem, HA2, or2, carry-in, s, c
  update procedure:
     update carry-in
     update half-adder-subsystem
     update HA2
     update or2
     update s
     update c

subsystem half-adder-subsystem is
  controls: x, y, not-x, not-y, and1, and2, and3, or1
  update procedure:
     update x
     update not-x
     update y
     update not-y
     update and1
     update and2
     update or1
     update and3
```

C.2.3 System/User Dialogue

```
.> (#> application definition test-full-adder)
application definition TEST-FULL-ADDER
  X Y CARRY-IN NOT-X NOT-Y AND1 AND2 AND3 OR1 OR2 S C HA2
    FULL-ADDER-TEST FULL-ADDER HALF-ADDER-SUBSYSTEM
.> (rs)
- Rules for: application definition TEST-FULL-ADDER
  X Y CARRY-IN NOT-X NOT-Y AND1 AND2 AND3 OR1 OR2 S C HA2
    FULL-ADDER-TEST FULL-ADDER HALF-ADDER-SUBSYSTEM -
2) CHECK-SEMANTICS
.> (ar 2)

More than one export can provide the data for IN1
                      which is used by object C
```

```
                                in subsystem FULL-ADDER
Choose the export item (subsystem and component)
  that you wish to be the source of this data:
      1> subsystem "FULL-ADDER" component "HA2" name "S"
      2> subsystem "FULL-ADDER" component "HA2" name "C"
      3> subsystem "FULL-ADDER" component "OR2" name "OUT1"
      4> subsystem "FULL-ADDER" component "CARRY-IN" name "OUT1"
      5> subsystem "HALF-ADDER-SUBSYSTEM" component "X" name "OUT1"
      6> subsystem "HALF-ADDER-SUBSYSTEM" component "Y" name "OUT1"
      7> subsystem "HALF-ADDER-SUBSYSTEM" component "NOT-X" name "OUT1"
      8> subsystem "HALF-ADDER-SUBSYSTEM" component "NOT-Y" name "OUT1"
      9> subsystem "HALF-ADDER-SUBSYSTEM" component "AND1" name "OUT1"
      10> subsystem "HALF-ADDER-SUBSYSTEM" component "AND2" name "OUT1"
      11> subsystem "HALF-ADDER-SUBSYSTEM" component "AND3" name "OUT1"
      12> subsystem "HALF-ADDER-SUBSYSTEM" component "OR1" name "OUT1"
      13> Specific source not required; use arbitrary one
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                    which is used by object S
                                in subsystem FULL-ADDER
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN2
                    which is used by object OR2
                                in subsystem FULL-ADDER
11

More than one export can provide the data for IN1
                    which is used by object OR2
                                in subsystem FULL-ADDER
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN2
                    which is used by object HA2
                                in subsystem FULL-ADDER
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN1
                    which is used by object HA2
                                in subsystem FULL-ADDER
Enter the number corresponding to the source you want to use
12

More than one export can provide the data for IN2
                    which is used by object OR1
                                in subsystem HALF-ADDER-SUBSYSTEM
```

```
Enter the number corresponding to the source you want to use
10

More than one export can provide the data for IN1
                    which is used by object OR1
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
9

More than one export can provide the data for IN2
                    which is used by object AND3
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for IN1
                    which is used by object AND3
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN2
                    which is used by object AND2
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for IN1
                    which is used by object AND2
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
7

More than one export can provide the data for IN2
                    which is used by object AND1
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
8

More than one export can provide the data for IN1
                    which is used by object AND1
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN1
                    which is used by object NOT-Y
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
6
```

```
More than one export can provide the data for IN1
                      which is used by object NOT-X
                                in subsystem HALF-ADDER-SUBSYSTEM
Enter the number corresponding to the source you want to use
5
Rule successfully applied.
application definition TEST-FULL-ADDER
  X Y CARRY-IN NOT-X NOT-Y AND1 AND2 AND3 OR1 OR2 S C HA2
    FULL-ADDER-TEST FULL-ADDER HALF-ADDER-SUBSYSTEM
.> (rs)
- Rules for: application definition TEST-FULL-ADDER
  X Y CARRY-IN NOT-X NOT-Y AND1 AND2 AND3 OR1 OR2 S C HA2
    FULL-ADDER-TEST FULL-ADDER HALF-ADDER-SUBSYSTEM -
1) DO-EXECUTE
2) CHECK-SEMANTICS
.> (ar 1)
LED S = OFF
LED C = ON
Rule successfully applied.
application definition TEST-FULL-ADDER
  X Y CARRY-IN NOT-X NOT-Y AND1 AND2 AND3 OR1 OR2 S C HA2
    FULL-ADDER-TEST FULL-ADDER HALF-ADDER-SUBSYSTEM
```

C.3 BCD Adder

This test case constructs a circuit which can be used to add two one-digit decimal numbers. Note that one-digit decimal number can range from 0 - 9; therefore four binary bits are needed for its computer representation which is called Binary Coded Decimal or BCD. Table C.1 provides a comparison between BCD and binary number representations (27:250). It can be used to verify the results of the constructed circuit.

Table C.1. BCD/Binary Comparison

C	S_8	S_4	S_2	S_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

C.3.1 Circuit Diagram See Figure C.4 ,(27:252).

C.3.2 Application Specification

```
application definition test-BCD-adder

switch a0  position: on
switch a1  position: off
switch a2  position: off
switch a3  position: on
```

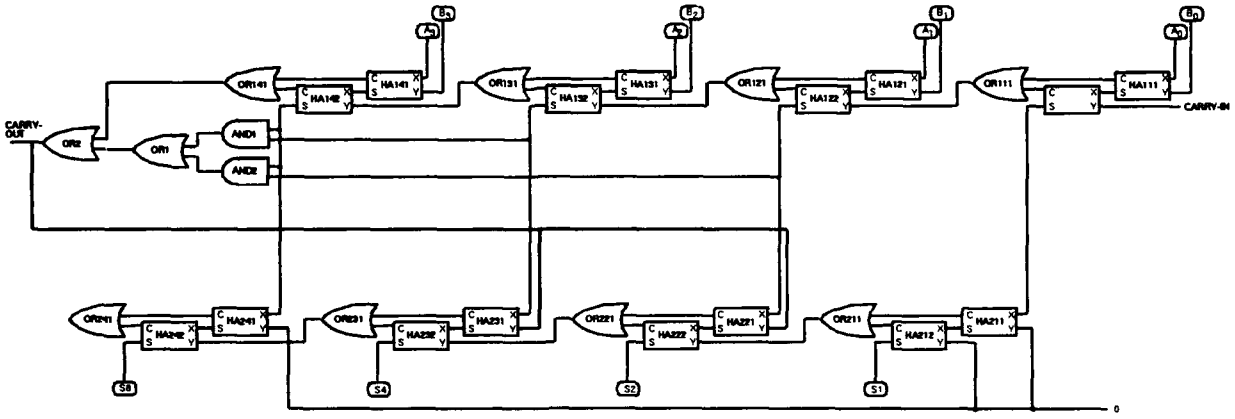

Figure C.4. BCD Adder

```
switch b0  position: on
switch b1  position: off
switch b2  position: off
switch b3  position: on

switch carry-in  position: off

switch zero position: off

half-adder HA111
half-adder HA112
half-adder HA121
half-adder HA122
half-adder HA131
half-adder HA132
half-adder HA141
half-adder HA142

half-adder HA211
half-adder HA212
half-adder HA221
half-adder HA222
half-adder HA231
half-adder HA232
half-adder HA241
half-adder HA242

or-gate or111
or-gate or121
or-gate or131
or-gate or141
```

```
or-gate or211
or-gate or221
or-gate or231
or-gate or241

or-gate or1
or-gate or2

and-gate and1
and-gate and2

led s1
led s2
led s4
led s8
led carry-out

application BCD-adder-test is
   controls: BCD-adder
   update procedure:
      update BCD-adder

subsystem BCD-adder is
  controls: four-bit-adder1, four-bit-adder2,
            and1, and2, or1, or2,
            a0, a1, a2, a3, b0, b1, b2, b3, carry-in, zero,
            s1, s2, s4, s8, carry-out
  update procedure:
     update a0
     update a1
     update a2
     update a3
     update b0
     update b1
     update b2
     update b3
     update carry-in
     update four-bit-adder1
     update and1
     update and2
     update or1
     update or2
     update zero
     update four-bit-adder2
     update s1
     update s2
     update s4
     update s8
     update carry-out
```

```
subsystem four-bit-adder1 is
  controls: full-adder111, full-adder121, full-adder131, full-adder141
  update procedure:
     update full-adder111
     update full-adder121
     update full-adder131
     update full-adder141

subsystem full-adder111 is
  controls:   HA111, HA112, or111
  update procedure:
     update HA111
     update HA112
     update or111

subsystem full-adder121 is
  controls:   HA121, HA122, or121
  update procedure:
     update HA121
     update HA122
     update or121

subsystem full-adder131 is
  controls:   HA131, HA132, or131
  update procedure:
     update HA131
     update HA132
     update or131


subsystem full-adder141 is
  controls:   HA141, HA142, or141
  update procedure:
     update HA141
     update HA142
     update or141


subsystem four-bit-adder2 is
  controls: full-adder211, full-adder221, full-adder231, full-adder241
  update procedure:
     update full-adder211
     update full-adder221
     update full-adder231
     update full-adder241

subsystem full-adder211 is
  controls:   HA211, HA212, or211
  update procedure:
     update HA211
     update HA212
```

```
      update or211


subsystem full-adder221 is
  controls:  HA221, HA222, or221
  update procedure:
     update HA221
     update HA222
     update or221

subsystem full-adder231 is
  controls:   HA231, HA232, or231
  update procedure:
     update HA231
     update HA232
     update or231

subsystem full-adder241 is
  controls:  HA241, HA242, or241
  update procedure:
     update HA241
     update HA242
     update or241
```

C.3.3 System/User Dialogue

```
.> (#> application definition test-BCD-adder)
application definition TEST-BCD-ADDER
  A0 A1 A2 A3 B0 B1 B2 B3 CARRY-IN ZERO HA111 HA112 HA121
    HA122 HA131 HA132 HA141 HA142 HA211 HA212 HA221 HA222 HA231
    HA232 HA241 HA242 OR111 OR121 OR131 OR141 OR211 OR221 OR231
    OR241 OR1 OR2 AND1 AND2 S1 S2 S4 S8 CARRY-OUT
    BCD-ADDER-TEST BCD-ADDER FOUR-BIT-ADDER1 FULL-ADDER111
    FULL-ADDER121 FULL-ADDER131 FULL-ADDER141 FOUR-BIT-ADDER2
    FULL-ADDER211 FULL-ADDER221 FULL-ADDER231 FULL-ADDER241
.> (rs)
- Rules for: application definition TEST-BCD-ADDER
  A0 A1 A2 A3 B0 B1 B2 B3 CARRY-IN ZERO HA111 HA112 HA121
    HA122 HA131 HA132 HA141 HA142 HA211 HA212 HA221 HA222 HA231
    HA232 HA241 HA242 OR111 OR121 OR131 OR141 OR211 OR221 OR231
    OR241 OR1 OR2 AND1 AND2 S1 S2 S4 S8 CARRY-OUT
    BCD-ADDER-TEST BCD-ADDER FOUR-BIT-ADDER1 FULL-ADDER111
    FULL-ADDER121 FULL-ADDER131 FULL-ADDER141 FOUR-BIT-ADDER2
    FULL-ADDER211 FULL-ADDER221 FULL-ADDER231 FULL-ADDER241 -
2) CHECK-SEMANTICS
.> (ar 2)

More than one export can provide the data for IN1
                    which is used by object CARRY-OUT
                              in subsystem BCD-ADDER
```

```
Choose the export item (subsystem and component)
  that you wish to be the source of this data:
      1> subsystem "BCD-ADDER" component "AND1" name "OUT1"
      2> subsystem "BCD-ADDER" component "AND2" name "OUT1"
      3> subsystem "BCD-ADDER" component "OR1" name "OUT1"
      4> subsystem "BCD-ADDER" component "OR2" name "OUT1"
      5> subsystem "BCD-ADDER" component "A0" name "OUT1"
      6> subsystem "BCD-ADDER" component "A1" name "OUT1"
      7> subsystem "BCD-ADDER" component "A2" name "OUT1"
      8> subsystem "BCD-ADDER" component "A3" name "OUT1"
      9> subsystem "BCD-ADDER" component "B0" name "OUT1"
     10> subsystem "BCD-ADDER" component "B1" name "OUT1"
     11> subsystem "BCD-ADDER" component "B2" name "OUT1"
     12> subsystem "BCD-ADDER" component "B3" name "OUT1"
     13> subsystem "BCD-ADDER" component "CARRY-IN" name "OUT1"
     14> subsystem "BCD-ADDER" component "ZERO" name "OUT1"
     15> subsystem "FULL-ADDER111" component "HA111" name "S"
     16> subsystem "FULL-ADDER111" component "HA111" name "C"
     17> subsystem "FULL-ADDER111" component "HA112" name "S"
     18> subsystem "FULL-ADDER111" component "HA112" name "C"
     19> subsystem "FULL-ADDER111" component "OR111" name "OUT1"
     20> subsystem "FULL-ADDER121" component "HA121" name "S"
     21> subsystem "FULL-ADDER121" component "HA121" name "C"
     22> subsystem "FULL-ADDER121" component "HA122" name "S"
     23> subsystem "FULL-ADDER121" component "HA122" name "C"
     24> subsystem "FULL-ADDER121" component "OR121" name "OUT1"
     25> subsystem "FULL-ADDER131" component "HA131" name "S"
     26> subsystem "FULL-ADDER131" component "HA131" name "C"
     27> subsystem "FULL-ADDER131" component "HA132" name "S"
     28> subsystem "FULL-ADDER131" component "HA132" name "C"
     29> subsystem "FULL-ADDER131" component "OR131" name "OUT1"
     30> subsystem "FULL-ADDER141" component "HA141" name "S"
     31> subsystem "FULL-ADDER141" component "HA141" name "C"
     32> subsystem "FULL-ADDER141" component "HA142" name "S"
     33> subsystem "FULL-ADDER141" component "HA142" name "C"
     34> subsystem "FULL-ADDER141" component "OR141" name "OUT1"
     35> subsystem "FULL-ADDER211" component "HA211" name "S"
     36> subsystem "FULL-ADDER211" component "HA211" name "C"
     37> subsystem "FULL-ADDER211" component "HA212" name "S"
     38> subsystem "FULL-ADDER211" component "HA212" name "C"
     39> subsystem "FULL-ADDER211" component "OR211" name "OUT1"
     40> subsystem "FULL-ADDER221" component "HA221" name "S"
     41> subsystem "FULL-ADDER221" component "HA221" name "C"
     42> subsystem "FULL-ADDER221" component "HA222" name "S"
     43> subsystem "FULL-ADDER221" component "HA222" name "C"
     44> subsystem "FULL-ADDER221" component "OR221" name "OUT1"
     45> subsystem "FULL-ADDER231" component "HA231" name "S"
     46> subsystem "FULL-ADDER231" component "HA231" name "C"
     47> subsystem "FULL-ADDER231" component "HA232" name "S"
     48> subsystem "FULL-ADDER231" component "HA232" name "C"
     49> subsystem "FULL-ADDER231" component "OR231" name "OUT1"
```

```
      50> subsystem "FULL-ADDER241" component "HA241" name "S"
      51> subsystem "FULL-ADDER241" component "HA241" name "C"
      52> subsystem "FULL-ADDER241" component "HA242" name "S"
      53> subsystem "FULL-ADDER241" component "HA242" name "C"
      54> subsystem "FULL-ADDER241" component "OR241" name "OUT1"
      55> Specific source not required; use arbitrary one
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN1
                      which is used by object S8
                                 in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
52

More than one export can provide the data for IN1
                      which is used by object S4
                                 in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
47

More than one export can provide the data for IN1
                      which is used by object S2
                                 in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
42

More than one export can provide the data for IN1
                      which is used by object S1
                                 in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
37

More than one export can provide the data for IN2
                      which is used by object OR2
                                 in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                      which is used by object OR2
                                 in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
34

More than one export can provide the data for IN2
                      which is used by object OR1
                                 in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
2
```

```
More than one export can provide the data for IN1
                    which is used by object OR1
                                in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN2
                    which is used by object AND2
                                in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
22

More than one export can provide the data for IN1
                    which is used by object AND2
                                in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
32

More than one export can provide the data for IN2
                    which is used by object AND1
                                in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
27

More than one export can provide the data for IN1
                    which is used by object AND1
                                in subsystem BCD-ADDER
Enter the number corresponding to the source you want to use
32

More than one export can provide the data for IN2
                    which is used by object OR111
                                in subsystem FULL-ADDER111
Enter the number corresponding to the source you want to use
18

More than one export can provide the data for IN1
                    which is used by object OR111
                                in subsystem FULL-ADDER111
Enter the number corresponding to the source you want to use
16

More than one export can provide the data for IN2
                    which is used by object HA112
                                in subsystem FULL-ADDER111
Enter the number corresponding to the source you want to use
13

More than one export can provide the data for IN1
                    which is used by object HA112
                                in subsystem FULL-ADDER111
```

```
Enter the number corresponding to the source you want to use
15

More than one export can provide the data for IN2
                    which is used by object HA111
                                in subsystem FULL-ADDER111
Enter the number corresponding to the source you want to use
9

More than one export can provide the data for IN1
                    which is used by object HA111
                                in subsystem FULL-ADDER111
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN2
                    which is used by object OR121
                                in subsystem FULL-ADDER121
Enter the number corresponding to the source you want to use
23

More than one export can provide the data for IN1
                    which is used by object OR121
                                in subsystem FULL-ADDER121
Enter the number corresponding to the source you want to use
21

More than one export can provide the data for IN2
                    which is used by object HA122
                                in subsystem FULL-ADDER121
Enter the number corresponding to the source you want to use
19

More than one export can provide the data for IN1
                    which is used by object HA122
                                in subsystem FULL-ADDER121
Enter the number corresponding to the source you want to use
20

More than one export can provide the data for IN2
                    which is used by object HA121
                                in subsystem FULL-ADDER121
Enter the number corresponding to the source you want to use
10

More than one export can provide the data for IN1
                    which is used by object HA121
                                in subsystem FULL-ADDER121
Enter the number corresponding to the source you want to use
6
```

```
More than one export can provide the data for IN2
                    which is used by object OR131
                                in subsystem FULL-ADDER131
Enter the number corresponding to the source you want to use
28

More than one export can provide the data for IN1
                    which is used by object OR131
                                in subsystem FULL-ADDER131
Enter the number corresponding to the source you want to use
26

More than one export can provide the data for IN2
                    which is used by object HA132
                                in subsystem FULL-ADDER131
Enter the number corresponding to the source you want to use
24

More than one export can provide the data for IN1
                    which is used by object HA132
                                in subsystem FULL-ADDER131
Enter the number corresponding to the source you want to use
25

More than one export can provide the data for IN2
                    which is used by object HA131
                                in subsystem FULL-ADDER131
Enter the number corresponding to the source you want to use
11

More than one export can provide the data for IN1
                    which is used by object HA131
                                in subsystem FULL-ADDER131
Enter the number corresponding to the source you want to use
7

More than one export can provide the data for IN2
                    which is used by object OR141
                                in subsystem FULL-ADDER141
Enter the number corresponding to the source you want to use
33

More than one export can provide the data for IN1
                    which is used by object OR141
                                in subsystem FULL-ADDER141
Enter the number corresponding to the source you want to use
31

More than one export can provide the data for IN2
                    which is used by object HA142
                                in subsystem FULL-ADDER141
```

```
Enter the number corresponding to the source you want to use
29

More than one export can provide the data for IN1
                    which is used by object HA142
                                in subsystem FULL-ADDER141
Enter the number corresponding to the source you want to use
30

More than one export can provide the data for IN2
                    which is used by object HA141
                                in subsystem FULL-ADDER141
Enter the number corresponding to the source you want to use
12

More than one export can provide the data for IN1
                    which is used by object HA141
                                in subsystem FULL-ADDER141
Enter the number corresponding to the source you want to use
8

More than one export can provide the data for IN2
                    which is used by object OR211
                                in subsystem FULL-ADDER211
Enter the number corresponding to the source you want to use
38

More than one export can provide the data for IN1
                    which is used by object OR211
                                in subsystem FULL-ADDER211
Enter the number corresponding to the source you want to use
36

More than one export can provide the data for IN2
                    which is used by object HA212
                                in subsystem FULL-ADDER211
Enter the number corresponding to the source you want to use
14

More than one export can provide the data for IN1
                    which is used by object HA212
                                in subsystem FULL-ADDER211
Enter the number corresponding to the source you want to use
35

More than one export can provide the data for IN2
                    which is used by object HA211
                                in subsystem FULL-ADDER211
Enter the number corresponding to the source you want to use
14
```

```
More than one export can provide the data for IN1
                    which is used by object HA211
                                in subsystem FULL-ADDER211
Enter the number corresponding to the source you want to use
17

More than one export can provide the data for IN2
                    which is used by object OR221
                                in subsystem FULL-ADDER221
Enter the number corresponding to the source you want to use
43

More than one export can provide the data for IN1
                    which is used by object OR221
                                in subsystem FULL-ADDER221
Enter the number corresponding to the source you want to use
41

More than one export can provide the data for IN2
                    which is used by object HA222
                                in subsystem FULL-ADDER221
Enter the number corresponding to the source you want to use
39

More than one export can provide the data for IN1
                    which is used by object HA222
                                in subsystem FULL-ADDER221
Enter the number corresponding to the source you want to use
40

More than one export can provide the data for IN2
                    which is used by object HA221
                                in subsystem FULL-ADDER221
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN1
                    which is used by object HA221
                                in subsystem FULL-ADDER221
Enter the number corresponding to the source you want to use
22

More than one export can provide the data for IN2
                    which is used by object OR231
                                in subsystem FULL-ADDER231
Enter the number corresponding to the source you want to use
48

More than one export can provide the data for IN1
                    which is used by object OR231
                                in subsystem FULL-ADDER231
```

```
Enter the number corresponding to the source you want to use
46

More than one export can provide the data for IN2
                    which is used by object HA232
                                in subsystem FULL-ADDER231
Enter the number corresponding to the source you want to use
44

More than one export can provide the data for IN1
                    which is used by object HA232
                                in subsystem FULL-ADDER231
Enter the number corresponding to the source you want to use
45

More than one export can provide the data for IN2
                    which is used by object HA231
                                in subsystem FULL-ADDER231
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN1
                    which is used by object HA231
                                in subsystem FULL-ADDER231
Enter the number corresponding to the source you want to use
27

More than one export can provide the data for IN2
                    which is used by object OR241
                                in subsystem FULL-ADDER241
Enter the number corresponding to the source you want to use
53

More than one export can provide the data for IN1
                    which is used by object OR241
                                in subsystem FULL-ADDER241
Enter the number corresponding to the source you want to use
51

More than one export can provide the data for IN2
                    which is used by object HA242
                                in subsystem FULL-ADDER241
Enter the number corresponding to the source you want to use
49

More than one export can provide the data for IN1
                    which is used by object HA242
                                in subsystem FULL-ADDER241
Enter the number corresponding to the source you want to use
50
```

```
More than one export can provide the data for IN2
                      which is used by object HA241
                                  in subsystem FULL-ADDER241
Enter the number corresponding to the source you want to use
14

More than one export can provide the data for IN1
                      which is used by object HA241
                                  in subsystem FULL-ADDER241
Enter the number corresponding to the source you want to use
32
Rule successfully applied.
application definition TEST-BCD-ADDER
  A0 A1 A2 A3 B0 B1 B2 B3 CARRY-IN ZERO HA111 HA112 HA121
    HA122 HA131 HA132 HA141 HA142 HA211 HA212 HA221 HA222 HA231
    HA232 HA241 HA242 OR111 OR121 OR131 OR141 OR211 OR221 OR231
    OR241 OR1 OR2 AND1 AND2 S1 S2 S4 S8 CARRY-OUT
    BCD-ADDER-TEST BCD-ADDER FOUR-BIT-ADDER1 FULL-ADDER111
    FULL-ADDER121 FULL-ADDER131 FULL-ADDER141 FOUR-BIT-ADDER2
    FULL-ADDER211 FULL-ADDER221 FULL-ADDER231 FULL-ADDER241
.> (rs)
- Rules for: application definition TEST-BCD-ADDER
  A0 A1 A2 A3 B0 B1 B2 B3 CARRY-IN ZERO HA111 HA112 HA121
    HA122 HA131 HA132 HA141 HA142 HA211 HA212 HA221 HA222 HA231
    HA232 HA241 HA242 OR111 OR121 OR131 OR141 OR211 OR221 OR231
    OR241 OR1 OR2 AND1 AND2 S1 S2 S4 S8 CARRY-OUT
    BCD-ADDER-TEST BCD-ADDER FOUR-BIT-ADDER1 FULL-ADDER111
    FULL-ADDER121 FULL-ADDER131 FULL-ADDER141 FOUR-BIT-ADDER2
    FULL-ADDER211 FULL-ADDER221 FULL-ADDER231 FULL-ADDER241 -
1) DO-EXECUTE
2) CHECK-SEMANTICS
.> (ar 1)
LED S1 = OFF
LED S2 = OFF
LED S4 = OFF
LED S8 = ON
LED CARRY-OUT = ON
Rule successfully applied.
application definition TEST-BCD-ADDER
  A0 A1 A2 A3 B0 B1 B2 B3 CARRY-IN ZERO HA111 HA112 HA121
    HA122 HA131 HA132 HA141 HA142 HA211 HA212 HA221 HA222 HA231
    HA232 HA241 HA242 OR111 OR121 OR131 OR141 OR211 OR221 OR231
    OR241 OR1 OR2 AND1 AND2 S1 S2 S4 S8 CARRY-OUT
    BCD-ADDER-TEST BCD-ADDER FOUR-BIT-ADDER1 FULL-ADDER111
    FULL-ADDER121 FULL-ADDER131 FULL-ADDER141 FOUR-BIT-ADDER2
    FULL-ADDER211 FULL-ADDER221 FULL-ADDER231 FULL-ADDER241
```

C.4 2 x 2 Binary Array Multiplier

This test case presents a circuit for multiplying two 2-digit binary numbers. It is based on the following formula (27:365):

```
              b1    b0
              a1    a0
             ----------
             a0b1  a0b0
       a1b1  a1b0
   --------------------
   c3    c2    c1    c0
```

C.4.1 Circuit Diagram

See Figure C.5 (27:365).

C.4.2 Application Specification

```
application definition test-2x2-binary-array-multiplier

switch a0  position: on
switch a1  position: on
switch b0  position: on
switch b1  position: on

and-gate and1
and-gate and2
and-gate and3
and-gate and4

half-adder HA1
half-adder HA2

led c0
led c1
led c2
led c3

application binary-array-test is
  controls: binary-array
  update procedure:
     update binary-array

subsystem binary-array is
  controls:  a0, a1, b0, b1,
```

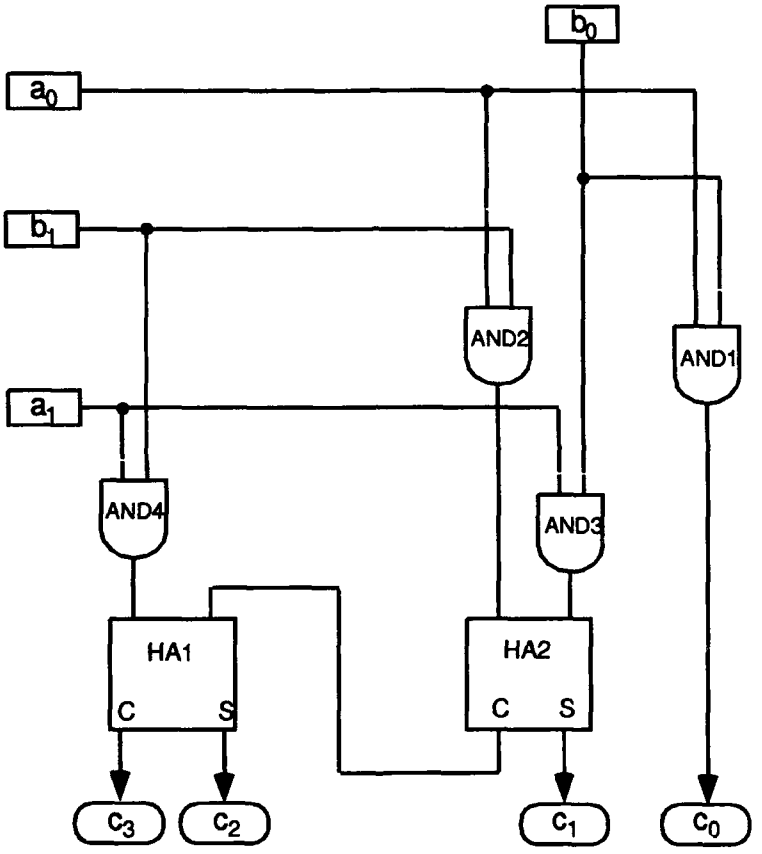

Figure C.5. 2 x 2 Binary Array Multiplier

```
            and1, and2, and3, and4,
            HA1, HA2,
            c0, c1, c2, c3
  update procedure:
     update a0
     update a1
     update b0
     update b1
     update and1
     update c0
     update and2
     update and3
     update HA1
     update c1
     update and4
     update HA2
     update c2
     update c3
```

C.4.3 System/User Dialogue

```
.> (#> application definition test-2x2-binary-array-multiplier)
application definition TEST-2X2-BINARY-ARRAY-MULTIPLIER
  A0 A1 B0 B1 AND1 AND2 AND3 AND4 HA1 HA2 C0 C1 C2 C3
    BINARY-ARRAY-TEST BINARY-ARRAY
.> (rs)
- Rules for: application definition TEST-2X2-BINARY-ARRAY-MULTIPLIER
  A0 A1 B0 B1 AND1 AND2 AND3 AND4 HA1 HA2 C0 C1 C2 C3
    BINARY-ARRAY-TEST BINARY-ARRAY -
2) CHECK-SEMANTICS
.> (ar 2)

More than one export can provide the data for IN1
                    which is used by object C3
                                in subsystem BINARY-ARRAY
Choose the export item (subsystem and component)
  that you wish to be the source of this data:
      1> subsystem "BINARY-ARRAY" component "A0" name "OUT1"
      2> subsystem "BINARY-ARRAY" component "A1" name "OUT1"
      3> subsystem "BINARY-ARRAY" component "B0" name "OUT1"
      4> subsystem "BINARY-ARRAY" component "B1" name "OUT1"
      5> subsystem "BINARY-ARRAY" component "AND1" name "OUT1"
      6> subsystem "BINARY-ARRAY" component "AND2" name "OUT1"
      7> subsystem "BINARY-ARRAY" component "AND3" name "OUT1"
      8> subsystem "BINARY-ARRAY" component "AND4" name "OUT1"
      9> subsystem "BINARY-ARRAY" component "HA1" name "S"
      10> subsystem "BINARY-ARRAY" component "HA1" name "C"
      11> subsystem "BINARY-ARRAY" component "HA2" name "S"
      12> subsystem "BINARY-ARRAY" component "HA2" name "C"
      13> Specific source not required; use arbitrary one
Enter the number corresponding to the source you want to use
```

```
12

More than one export can provide the data for IN1
                      which is used by object C2
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
11

More than one export can provide the data for IN1
                      which is used by object C1
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
9

More than one export can provide the data for IN1
                      which is used by object C0
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN2
                      which is used by object HA2
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
8

More than one export can provide the data for IN1
                      which is used by object HA2
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
10

More than one export can provide the data for IN2
                      which is used by object HA1
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for IN1
                      which is used by object HA1
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
7

More than one export can provide the data for IN2
                      which is used by object AND4
                                  in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN1
```

```
                    which is used by object AND4
                                in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN2
                    which is used by object AND3
                                in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                    which is used by object AND3
                                in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN2
                    which is used by object AND2
                                in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN1
                    which is used by object AND2
                                in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
1

More than one export can provide the data for IN2
                    which is used by object AND1
                                in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN1
                    which is used by object AND1
                                in subsystem BINARY-ARRAY
Enter the number corresponding to the source you want to use
1
Rule successfully applied.
application definition TEST-2X2-BINARY-ARRAY-MULTIPLIER
  A0 A1 B0 B1 AND1 AND2 AND3 AND4 HA1 HA2 C0 C1 C2 C3
    BINARY-ARRAY-TEST BINARY-ARRAY
.> (rs)
- Rules for: application definition TEST-2X2-BINARY-ARRAY-MULTIPLIER
  A0 A1 B0 B1 AND1 AND2 AND3 AND4 HA1 HA2 C0 C1 C2 C3
    BINARY-ARRAY-TEST BINARY-ARRAY -
1) DO-EXECUTE
2) CHECK-SEMANTICS
.> (ar 1)
```

```
LED C0 = ON
LED C1 = OFF
LED C2 = OFF
LED C3 = ON
Rule successfully applied.
application definition TEST-2X2-BINARY-ARRAY-MULTIPLIER
  A0 A1 B0 B1 AND1 AND2 AND3 AND4 HA1 HA2 C0 C1 C2 C3
    BINARY-ARRAY-TEST BINARY-ARRAY
```

C.5 Universal Shift Register

This test case builds a universal 4-bit shift register which, depending on the value of the select lines, can load 4 bits into the register, shift the contents of the register to the left, shift the contents to the right or do nothing. Often these registers are constructed using D flipflops. However, since the validating domain contains no D flipflops, JK flipflops were substituted. Table C.2 summarizes the expected action for various select line values.

Table C.2. Universal Shift Register Controls

s_1	s_0	Function
0	0	shift contents right
1	0	shift contents left
0	1	load input into register
1	1	do nothing

C.5.1 Circuit Diagram See Figure C.6 (8:287).

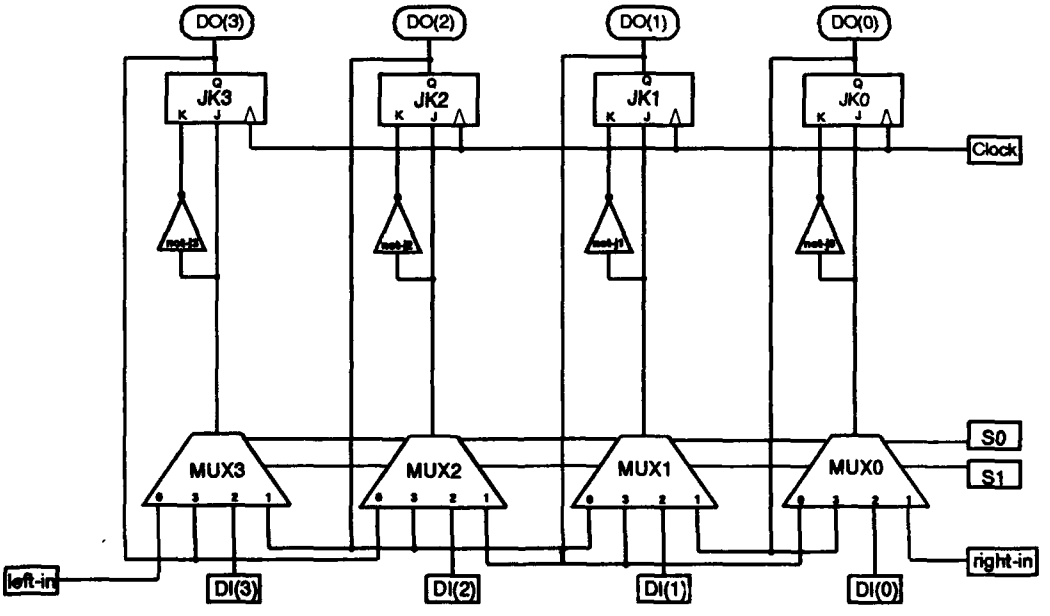

Figure C.6. Universal Shift Register

C.5.2 Application Specification

```
application definition test-universal-shift-register

switch di0 position: off
switch di1 position: on
switch di2 position: off
switch di3 position: on

switch s0  position: on
switch s1  position: off

switch left-in  position: off
switch right-in position: off
switch clock    position: on

led do0
led do1
led do2
led do3

mux mux0
mux mux1
mux mux2
mux mux3

not-gate not-j0
not-gate not-j1
not-gate not-j2
not-gate not-j3

jk-flip-flop jk0 state off
jk-flip-flop jk1 state off
jk-flip-flop jk2 state off
jk-flip-flop jk3 state off

application universal-shift-register is
  controls: universal-shift-reg
  update procedure:
       update universal-shift-reg
       update universal-shift-reg

subsystem universal-shift-reg is
  controls:  di0, di1, di2, di3, s0, s1, left-in, right-in, clock,
             do0, do1, do2, do3,
             mux0, mux1, mux2, mux3,
             jk0, jk1, jk2, jk3,
             not-j0, not-j1, not-j2, not-j3
  update procedure:
     update di0
     update di1
     update di2
     update di3
```

```
    update right-in
    update left-in
    update clock
    update s0
    update s1
    if s1.out1 and not s0.out1 then
      update mux3
      update not-j3
      update jk3
      update mux2
      update not-j2
      update jk2
      update mux1
      update not-j1
      update jk1
      update mux0
      update not-j0
      update jk0
      update do3
      update do2
      update do1
      update do0
    else
      update mux0
      update not-j0
      update jk0
      update mux1
      update not-j1
      update jk1
      update mux2
      update not-j2
      update jk2
      update mux3
      update not-j3
      update jk3
      update do3
      update do2
      update do1
      update do0
    end if
  setstate s0 (position, off)
  setstate s1 (position, off)
```

C.5.3 System/User Dialogue

```
.> (#> application definition test-universal-shift-register)
application definition TEST-UNIVERSAL-SHIFT-REGISTER
  DI0 DI1 DI2 DI3 S0 S1 LEFT-IN RIGHT-IN CLOCK DO0 DO1 DO2
    DO3 MUX0 MUX1 MUX2 MUX3 NOT-J0 NOT-J1 NOT-J2 NOT-J3 JK0 JK1
    JK2 JK3 UNIVERSAL-SHIFT-REGISTER UNIVERSAL-SHIFT-REG
.> (rs)
```

```
- Rules for: application definition TEST-UNIVERSAL-SHIFT-REGISTER
  DI0 DI1 DI2 DI3 S0 S1 LEFT-IN RIGHT-IN CLOCK DO0 DO1 DO2
    DO3 MUX0 MUX1 MUX2 MUX3 NOT-J0 NOT-J1 NOT-J2 NOT-J3 JK0 JK1
    JK2 JK3 UNIVERSAL-SHIFT-REGISTER UNIVERSAL-SHIFT-REG -
2) CHECK-SEMANTICS
.> (ar 2)
There is more than one possibility for data with name OUT1
  Choose the data you would like to use for evaluating the conditional
      1> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = NOT-J3: name = OUT1
      2> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = NOT-J2: name = OUT1
      3> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = NOT-J1: name = OUT1
      4> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = NOT-J0: name = OUT1
      5> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = MUX3: name = OUT1
      6> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = MUX2: name = OUT1
      7> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = MUX1: name = OUT1
      8> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = MUX0: name = OUT1
      9> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = CLOCK: name = OUT1
      10> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = RIGHT-IN: name = OUT1
      11> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = LEFT-IN: name = OUT1
      12> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = S1: name = OUT1
      13> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = S0: name = OUT1
      14> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = DI3: name = OUT1
      15> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = DI2: name = OUT1
      16> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = DI1: name = OUT1
      17> In export area of subsystem UNIVERSAL-SHIFT-REG:
producer = DI0: name = OUT1
12

More than one export can provide the data for IN1
                    which is used by object NOT-J3
                                in subsystem UNIVERSAL-SHIFT-REG
Choose the export item (subsystem and component)
  that you wish to be the source of this data:
      1> subsystem "UNIVERSAL-SHIFT-REG" component "DI0" name "OUT1"
      2> subsystem "UNIVERSAL-SHIFT-REG" component "DI1" name "OUT1"
```

```
      3> subsystem "UNIVERSAL-SHIFT-REG" component "DI2" name "OUT1"
      4> subsystem "UNIVERSAL-SHIFT-REG" component "DI3" name "OUT1"
      5> subsystem "UNIVERSAL-SHIFT-REG" component "S0" name "OUT1"
      6> subsystem "UNIVERSAL-SHIFT-REG" component "S1" name "OUT1"
      7> subsystem "UNIVERSAL-SHIFT-REG" component "LEFT-IN" name "OUT1"
      8> subsystem "UNIVERSAL-SHIFT-REG" component "RIGHT-IN" name "OUT1"
      9> subsystem "UNIVERSAL-SHIFT-REG" component "CLOCK" name "OUT1"
      10> subsystem "UNIVERSAL-SHIFT-REG" component "MUX0" name "OUT1"
      11> subsystem "UNIVERSAL-SHIFT-REG" component "MUX1" name "OUT1"
      12> subsystem "UNIVERSAL-SHIFT-REG" component "MUX2" name "OUT1"
      13> subsystem "UNIVERSAL-SHIFT-REG" component "MUX3" name "OUT1"
      14> subsystem "UNIVERSAL-SHIFT-REG" component "JK0" name "Q"
      15> subsystem "UNIVERSAL-SHIFT-REG" component "JK0" name "Q-BAR"
      16> subsystem "UNIVERSAL-SHIFT-REG" component "JK1" name "Q"
      17> subsystem "UNIVERSAL-SHIFT-REG" component "JK1" name "Q-BAR"
      18> subsystem "UNIVERSAL-SHIFT-REG" component "JK2" name "Q"
      19> subsystem "UNIVERSAL-SHIFT-REG" component "JK2" name "Q-BAR"
      20> subsystem "UNIVERSAL-SHIFT-REG" component "JK3" name "Q"
      21> subsystem "UNIVERSAL-SHIFT-REG" component "JK3" name "Q-BAR"
      22> subsystem "UNIVERSAL-SHIFT-REG" component "NOT-J0" name "OUT1"
      23> subsystem "UNIVERSAL-SHIFT-REG" component "NOT-J1" name "OUT1"
      24> subsystem "UNIVERSAL-SHIFT-REG" component "NOT-J2" name "OUT1"
      25> subsystem "UNIVERSAL-SHIFT-REG" component "NOT-J3" name "OUT1"
      26> Specific source not required; use arbitrary one
Enter the number corresponding to the source you want to use
13

More than one export can provide the data for IN1
                    which is used by object NOT-J2
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
12

More than one export can provide the data for IN1
                    which is used by object NOT-J1
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
11

More than one export can provide the data for IN1
                    which is used by object NOT-J0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
10

More than one export can provide the data for CLK
                    which is used by object JK3
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
9
```

```
More than one export can provide the data for K
                      which is used by object JK3
                                 in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
25

More than one export can provide the data for J
                      which is used by object JK3
                                 in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
13

More than one export can provide the data for CLK
                      which is used by object JK2
                                 in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
9

More than one export can provide the data for K
                      which is used by object JK2
                                 in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
24

More than one export can provide the data for J
                      which is used by object JK2
                                 in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
12

More than one export can provide the data for CLK
                      which is used by object JK1
                                 in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
9

More than one export can provide the data for K
                      which is used by object JK1
                                 in subsystem UNIVERSAL-SHIFT-REG

Enter the number corresponding to the source you want to use
23

More than one export can provide the data for J
                      which is used by object JK1
                                 in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
11

More than one export can provide the data for CLK
                      which is used by object JK0
```

```
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
9

More than one export can provide the data for K
                     which is used by object JK0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
22

More than one export can provide the data for J
                     which is used by object JK0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
10

More than one export can provide the data for S1
                     which is used by object MUX3
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for S0
                     which is used by object MUX3
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN3
                     which is used by object MUX3
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
20

More than one export can provide the data for IN2
                     which is used by object MUX3
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
18

More than one export can provide the data for IN1
                     which is used by object MUX3
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
4

More than one export can provide the data for IN0
                     which is used by object MUX3
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
7
```

```
More than one export can provide the data for S1
                      which is used by object MUX2
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for S0
                      which is used by object MUX2
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN3
                      which is used by object MUX2
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
18

More than one export can provide the data for IN2
                      which is used by object MUX2
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
16

More than one export can provide the data for IN1
                      which is used by object MUX2
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
3

More than one export can provide the data for IN0
                      which is used by object MUX2
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
20

More than one export can provide the data for S1
                      which is used by object MUX1
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for S0
                      which is used by object MUX1
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN3
                      which is used by object MUX1
```

```
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
16

More than one export can provide the data for IN2
                     which is used by object MUX1
                                in subsystem UNIVERSAL-SHIFT-REG
CEnter the number corresponding to the source you want to use
14

More than one export can provide the data for IN1
                     which is used by object MUX1
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
2

More than one export can provide the data for IN0
                     which is used by object MUX1
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
18

More than one export can provide the data for S1
                     which is used by object MUX0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
6

More than one export can provide the data for S0
                     which is used by object MUX0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
5

More than one export can provide the data for IN3
                     which is used by object MUX0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
14

More than one export can provide the data for IN2
                     which is used by object MUX0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
8

More than one export can provide the data for IN1
                     which is used by object MUX0
                                in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
1
```

```
More than one export can provide the data for IN0
                      which is used by object MUX0
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
16

More than one export can provide the data for IN1
                      which is used by object DO3
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
20

More than one export can provide the data for IN1
                      which is used by object DO2
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
18

More than one export can provide the data for IN1
                      which is used by object DO1
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
16

More than one export can provide the data for IN1
                      which is used by object DO0
                                  in subsystem UNIVERSAL-SHIFT-REG
Enter the number corresponding to the source you want to use
14
Rule successfully applied.
application definition TEST-UNIVERSAL-SHIFT-REGISTER
  DI0 DI1 DI2 DI3 S0 S1 LEFT-IN RIGHT-IN CLOCK DO0 DO1 DO2
    DO3 MUX0 MUX1 MUX2 MUX3 NOT-J0 NOT-J1 NOT-J2 NOT-J3 JK0 JK1
    JK2 JK3 UNIVERSAL-SHIFT-REGISTER UNIVERSAL-SHIFT-REG
.> (rs)
- Rules for: application definition TEST-UNIVERSAL-SHIFT-REGISTER
  DI0 DI1 DI2 DI3 S0 S1 LEFT-IN RIGHT-IN CLOCK DO0 DO1 DO2
    DO3 MUX0 MUX1 MUX2 MUX3 NOT-J0 NOT-J1 NOT-J2 NOT-J3 JK0 JK1
    JK2 JK3 UNIVERSAL-SHIFT-REGISTER UNIVERSAL-SHIFT-REG -
1) DO-EXECUTE
2) CHECK-SEMANTICS
.> (ar 1)
LED DO3 = ON
LED DO2 = OFF
LED DO1 = ON
LED DO0 = OFF
LED DO3 = OFF
LED DO2 = ON
LED DO1 = OFF
LED DO0 = ON
```

```
Rule successfully applied.
application definition TEST-UNIVERSAL-SHIFT-REGISTER
  DI0 DI1 DI2 DI3 S0 S1 LEFT-IN RIGHT-IN CLOCK DO0 DO1 DO2
    DO3 MUX0 MUX1 MUX2 MUX3 NOT-J0 NOT-J1 NOT-J2 NOT-J3 JK0 JK1
    JK2 JK3 UNIVERSAL-SHIFT-REGISTER UNIVERSAL-SHIFT-REG
```

Appendix D. *Code*

This appendix contains the REFINE source code for the *Preprocessing*, *Semantic Check* and *Execute* portions of Architect, the application composer described in Section 4.1. Each section corresponds to an individual source code file.

D.1 Globals Definitions

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: globals.re

 Description: Contains all the global constants and variables.

||#


constant  Saved-Suffix : string = "-SAVED"

constant Generics-Path : string = "../generics/"

constant Applic-Path : string = "../applics/"

constant Object-Path : string = "../objs/"

constant separator : char = #\.

var Fatal-Error : boolean = false

var Changes-Made : boolean = false

var Semantic-Checks-Performed : boolean = false
```

D.2 REFINE Domain Model

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: dm-ocu.re

 Description:
```

```
  This version includes only the domain-independent dm
  data.  Domain-specific domain knowledge is included in the
  technology base in the file for the corresponding object
  class.

||#

% OBJECT CLASSES:

var World-Obj                    : object-class subtype-of user-Object

  var Spec-Obj                   : object-class subtype-of World-Obj
    % A high-level object that ties together all of the parts of an
    % application definition

  var Spec-Part-Obj              : object-class subtype-of World-Obj
    % Spec-Parts describe all of the senetences used by the application
    % specialist to build an application definition

    var Incomplete-Obj           : object-class subtype-of Spec-Part-Obj
    var Generic-Inst             : object-class subtype-of Spec-Part-Obj
    var Load-Obj                 : object-class subtype-of Spec-Part-Obj

    var Component-Obj            : object-class subtype-of Spec-Part-Obj
      % Component-Obj's are all the parts of a final definition

      var Application-Obj        : object-class subtype-of Component-Obj
      var Subsystem-Obj          : object-class subtype-of Component-Obj
      var Primitive-Obj          : object-class subtype-of Component-Obj

  var Statement-Obj              : object-class subtype-of World-Obj
    var If-Stmt-Obj              : object-class subtype-of Statement-Obj
    var While-Stmt-Obj           : object-class subtype-of Statement-Obj
    var Call-Obj                 : object-class subtype-of Statement-Obj
      var Update-Call-Obj        : object-class subtype-of Call-Obj
      var Create-Call-Obj        : object-class subtype-of Call-Obj
      var SetFunction-Call-Obj   : object-class subtype-of Call-Obj
      var SetState-Call-Obj      : object-class subtype-of Call-Obj
      var Destroy-Call-Obj       : object-class subtype-of Call-Obj
      var Initialize-Call-Obj    : object-class subtype-of call-obj
      var Stabilize-Call-Obj     : object-class subtype-of call-obj
      var Configure-Call-Obj     : object-class subtype-of call-obj

  var Import-Export-Obj          : object-class subtype-of World-Obj
  var Import-Obj                 : object-class subtype-of Import-Export-Obj
  var Export-Obj                 : object-class subtype-of Import-Export-Obj
  var Name-Value-Obj             : object-class subtype-of World-Obj
  var Source-Obj                 : object-class subtype-of World-Obj

  var Generic-Obj                : object-class subtype-of World-Obj
```

```
%%% ATTRIBUTES:

% Spec-Obj:
var Spec-Parts    : map(Spec-Obj, seq(Spec-Part-Obj))
    computed-using
          Spec-Parts(x) = []

% Incomplete-Obj:
var Obj-Type      : map(Incomplete-Obj, symbol)     = {| |}


% Generic-Inst:
var Generic-To-Be-Used : map(Generic-Inst, symbol) = {| |}
var Generic-Parameters : map(Generic-Inst, seq(any-type))
  computed-using
    Generic-Parameters(x) = []

% Load-Obj:
var Object-To-Load      : map(Load-Obj, symbol)     = {| |}


% Application-Obj:
 var Application-Components    : map(Application-Obj, seq(symbol))
   computed-using
     Application-Components(x) = []

 var Application-update        : map(Application-Obj, seq(Statement-Obj))
   computed-using
     Application-update(x) = []

% Subsystem-Obj:
var Controllees               : map(Subsystem-Obj, seq(symbol))
   computed-using
     controllees(x) = []

%%    changed seq to set in import-Area and Export-Area ...
var Import-Area               : map(Subsystem-Obj, set(Import-Obj))
   computed-using
    Import-Area(x) = {}
var Export-Area               : map(Subsystem-Obj, set(Export-Obj))
   computed-using
     Export-Area(x) = {}

var Update                    : map(Subsystem-Obj, seq(Statement-Obj))
   computed-using
     Update(x) = []
var Initialize                : map(subsystem-obj, seq(name-value-obj))
   computed-using
     Initialize(x) = []

% Statements:
```

```
% If-Stmt-Obj

var If-Cond                  : map(If-stmt-Obj, Expression)        = {| |}
var Then-Stmts               : map(If-stmt-Obj, seq(Statement-Obj))
 computed-using
    Then-Stmts(x) = []
var Else-Stmts               : map(If-stmt-Obj, seq(Statement-Obj))
 computed-using
    Else-Stmts(x) = []

% While-Stmt-Obj:

var  While-cond               : map(While-stmt-Obj, expression)     = {| |}
var  While-stmts              : map(While-stmt-Obj, seq(Statement-Obj))
  computed-using
     While-stmts(x) = []

% Call-Obj:
var operand                  : map(Call-Obj, symbol) = {| |}

 % Create-Call-Obj:
 var object-type             : map(create-Call-Obj, symbol) = {||}

 % SetFunction-Call-Obj:
 var function-name           : map(SetFunction-Call-Obj, symbol) = {||}
 var coefficients            : map(SetFunction-Call-Obj, set(name-value-Obj))
   computed-using
     coefficients(x) = {}

 % SetState-Call-Obj:
 var state-changes           : map(SetState-Call-Obj, set(name-value-Obj))
   computed-using
      state-changes(x) = {}

% Import-Obj:
var import-name              : map(Import-Obj, symbol)          = {||}
var import-category          : map(Import-Obj, symbol)          = {||}
var import-type-data         : map(Import-Obj, symbol)          = {||}
var consumer                 : map(Import-Obj, symbol)          = {||}
var Source                   : map(Import-Obj, set(Source-Obj))
  computed-using
    Source(x) = {}

% Export-Obj:
var export-name              : map(Export-Obj, symbol)   = {||}
var export-category          : map(Export-Obj, symbol)   = {||}
var export-type-data         : map(Export-Obj, symbol)   = {||}
var value                    : map(Export-Obj, any-type) = {||}
var producer                 : map(Export-Obj, symbol)   = {||}
```

```
% Name-Value-Obj:
var Name-value-Name          : map(Name-value-Obj, symbol) = {||}
var Name-value-value         : map(Name-value-Obj, any-type) = {||}

% Source-Obj:
var Source-Subsystem        : map(Source-Obj, symbol)     = {||}
var Source-Object           : map(Source-Obj, symbol)     = {||}
var Source-Name             : map(Source-Obj, symbol)     = {||}

% Generic-Obj:
var Obj-Instance        : map(Generic-Obj, Symbol)   = {| |}
var Placeholder-IDs     : map(Generic-Obj, seq(any-type))
  computed-using
    Placeholder-IDs(x) = □

var Placeholder-Type    : map(Generic-Obj, seq(symbol))
  computed-using
    Placeholder-Type(x) = □

%-------------------------------------------------------------------------------
% Code for Boolean-expressions
 var Expression : object-class subtype-of World-Obj

 var Literal-Expression : object-class subtype-of expression

 var Identifier         : object-class subtype-of Literal-Expression
 var Boolean-Literal    : object-class subtype-of Literal-Expression
    var True-Literal    : object-class subtype-of Boolean-Literal
    var False-Literal   : object-class subtype-of Boolean-Literal
 var Number-Literal     : object-class subtype-of Literal-Expression
   var Integer-Literal  : object-class subtype-of Number-Literal
   var Real-Literal     : object-class subtype-of Number-Literal
 var String-Literal     : object-class subtype-of Literal-Expression


 var Unary-Expression : object-class subtype-of Expression
    var Not-Exp       : object-class subtype-of Unary-Expression
    var abs-exp       : object-class subtype-of unary-expression
    var negate-exp    : object-class subtype-of unary-expression
    var positive-exp  : object-class subtype-of unary-expression

 var Binary-Expression   : object-class subtype-of Expression
    var Or-Exp           : object-class subtype-of Binary-Expression
    var And-Exp          : object-class subtype-of Binary-Expression
    var Equal-Exp        : object-class subtype-of Binary-Expression
    var Not-Equal-Exp    : object-class subtype-of Binary-Expression
    var LT-Exp           : object-class subtype-of Binary-Expression
    var LTE-Exp          : object-class subtype-of Binary-Expression
    var GT-Exp           : object-class subtype-of Binary-Expression
    var GTE-Exp          : object-class subtype-of Binary-Expression
```

```
    var add-exp          : object-class subtype-of Binary-Expression
    var subtract-exp     : object-class subtype-of Binary-Expression
    var multiply-exp     : object-class subtype-of Binary-Expression
    var divide-exp       : object-class subtype-of Binary-Expression
    var mod-exp          : object-class subtype-of Binary-Expression
    var exponential-exp  : object-class subtype-of Binary-Expression


%%  Attributes for expressions:
 var Id-Name          : map(Identifier, symbol) = {| |}
 var Id-Source        : map(Identifier, import-export-obj) = {| |}

 var Int-value        : map(Integer-Literal, integer) = {| |}
 var Real-value       : map(Real-Literal, real) = {| |}
 var Boolean-value    : map(Boolean-Literal, boolean) = {| |}
 var String-value     : map(String-Literal, string) = {| |}

 var Argument1 : map(Binary-Expression, Expression) = {| |}
 var Argument2 : map(Binary-Expression, Expression) = {| |}
 var Argument  : map(Unary-Expression,  Expression) = {| |}

%% Tree Attributes For Expressions
Form Expression-Attrs
  define-tree-attributes('Binary-Expression, {'Argument1, 'Argument2});
  define-tree-attributes('Unary-Expression, {'Argument})


Form Define-AST
  define-tree-attributes('While-Stmt-Obj, {'While-Cond, 'While-Stmts});
  define-tree-attributes('If-Stmt-Obj, {'If-Cond, 'Then-Stmts, 'Else-Stmts});
  define-tree-attributes('Call-Obj, {'Operand});
  define-tree-attributes('SetFunction-Call-Obj, {'Function-Name,
                          'Coefficients});
  define-tree-attributes('SetState-Call-Obj, {'state-changes});
  define-tree-attributes('Application-Obj, {'Application-Components,
                          'Application-Update});
  define-tree-attributes('Subsystem-Obj,
          {'Controllees, 'Update, 'Initialize, 'Export-Area, 'Import-Area});
  define-tree-attributes('Spec-Obj, {'Spec-Parts});
  define-tree-attributes('Import-Obj, {'Import-Name, 'Import-Category,
                          'Import-Type-Data, 'Source, 'Consumer});
  define-tree-attributes('Export-Obj, {'Export-Name, 'Export-Category,
                          'Export-Type-Data, 'Value, 'Producer});
  define-tree-attributes('Generic-Obj, {'Obj-Instance, 'Placeholder-Ids})

 form Make-Names-Unique
  unique-names-class('Spec-Obj, true);
  unique-names-class('Application-Obj, true);
  unique-names-class('Subsystem-Obj, true);
  unique-names-class('Generic-Obj, true);
  unique-names-class('Generic-Inst, true);
```

```
  unique-names-class('Load-Obj, true);
  unique-names-class('Incomplete-Obj, true)
```

D.3 OCU Grammar

```
!! in-package("RU")
!! in-grammar('syntax)

#||
 File name:  gram-ocu.re

 Description: The OCU grammar - all of the domain-independent productions, most of which
  describe the OCU model
  NOTE: If you change this file, you must also recompile the domain-specific grammar.
        If no changes are made to that grammar, erase its .fasl4 file to make sure it
        recompiles.  Otherwise, you won't see the changes made to the OCU grammar.
        (See the DIALECT User's guide about grammar inheritance)
 Rules:
  None

 Functions:
  None

||#


grammar OCU

  no-patterns

  start-classes Spec-Obj, Subsystem-obj, Incomplete-obj, Load-Obj, Generic-Obj

  file-classes Spec-Obj, Subsystem-obj, Incomplete-obj, Load-Obj, Generic-Obj

  productions

    Spec-Obj        ::= ["application" "definition" name {Spec-Parts + ""} ]
                                                        builds Spec-Obj,

    Application-Obj      ::= ["application" name "is"
                              "controls:" application-components * ","
                               "update procedure:"
                               application-update * ""] builds Application-Obj,

    Subsystem-Obj     ::= ["subsystem" name "is"
                           "controls:" Controllees * ","
                           {["imports:" Import-Area * ""]}
                           {["exports:" Export-Area * ""]}
                           {["initialize procedure:"
```

```
                        initialize * ""]}
                      "update procedure:"
                       update * "" ]                    builds Subsystem-Obj,

Import-Obj       ::= [import-name import-category  import-type-data
                      consumer  "(" [source * ""] ")"]    builds Import-Obj,

Export-Obj       ::= [export-name export-category  export-type-data
                      value  producer]                 builds Export-Obj,

Source-Obj       ::= [Source-Name Source-Subsystem Source-Object]
                             builds Source-Obj,

Generic-Inst     ::= ["generic" name "is" "new" Generic-To-Be-Used
                      Generic-Parameters * ","]    builds Generic-Inst,

Incomplete-Obj   ::= ["object" obj-type "," name]  builds Incomplete-obj,

Load-Obj         ::= ["load" Object-To-Load]      builds Load-Obj,

If-Stmt-Obj      ::= ["if" if-cond "then" Then-Stmts + ""
                     {["else" Else-Stmts + ""]}
                     "end" "if" ]                 builds If-Stmt-Obj,

While-Stmt-Obj  ::= ["while" while-cond "loop"
                      While-Stmts * ""
                      "end"  "while" ]            builds While-Stmt-Obj,

Update-Call-Obj  ::= ["update" operand]           builds Update-Call-Obj,

Create-Call-Obj  ::= ["create" operand object-type] builds Create-Call-Obj,

SetFunction-Call-Obj ::= ["setfunction" operand function-name
                             Coefficients * ""] builds SetFunction-Call-Obj,

SetState-Call-Obj    ::= ["setstate" operand
                             State-Changes * ""] builds SetState-Call-Obj,

Destroy-Call-Obj     ::= ["destroy" operand]     builds Destroy-Call-Obj,

Initialize-Call-Obj  ::= ["initialize" operand]  builds Initialize-Call-Obj,

Configure-Call-Obj   ::= ["configure" operand]   builds Configure-Call-Obj,

Stabilize-Call-Obj   ::= ["stabilize" operand]   builds Stabilize-Call-Obj,

Name-Value-Obj       ::= ["(" name-value-name ","
                             name-value-value ")"]   builds Name-Value-Obj,


Generic-Obj      ::= ["generic-obj" name Obj-Instance
```

```
                    "ids:"  {Placeholder-IDs + ","}
                    "types:" {Placeholder-Type + ","}]  builds Generic-Obj,

And-Exp ::= [argument1 "and" argument2]      builds And-Exp,
Or-Exp  ::= [argument1 "or"  argument2]      builds or-Exp,

Equal-Exp     ::= [argument1 "="  argument2] builds Equal-Exp,
Not-Equal-Exp ::= [argument1 "/=" argument2] builds Not-Equal-Exp,
LT-Exp        ::= [argument1 "<"  argument2] builds LT-Exp,
LTE-Exp       ::= [argument1 "<=" argument2] builds LTE-Exp,
GT-Exp        ::= [argument1 ">"  argument2] builds GT-Exp,
GTE-Exp       ::= [argument1 ">=" argument2] builds GTE-Exp,

Add-Exp       ::= [argument1 "+" argument2]  builds Add-Exp,
Subtract-Exp  ::= [argument1 "-" argument2]  builds Subtract-Exp,

Multiply-Exp    ::= [argument1 "*"   argument2] builds Multiply-Exp,
Divide-Exp      ::= [argument1 "/"   argument2] builds Divide-Exp,
Mod-Exp         ::= [argument1 "mod" argument2] builds Mod-Exp,
Exponential-Exp ::= [argument1 "**"  argument2] builds Exponential-Exp,

Abs-Exp      ::= ["abs" argument] builds Abs-Exp,
Not-Exp      ::= ["not" argument] builds Not-Exp,
Negate-Exp   ::= ["-" argument]   builds Negate-Exp,
Positive-Exp ::= ["+" argument]   builds Positive-Exp,

Identifier      ::= [Id-Name]      builds Identifier,
Integer-Literal ::= [Int-Value]    builds Integer-Literal,
Real-Literal    ::= [Real-Value]   builds Real-Literal,
String-Literal  ::= [String-Value] builds String-Literal,
True-Literal    ::= ["true"]       builds True-Literal,
False-Literal   ::= ["false"]      builds False-Literal

symbol-start-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ."

symbol-continue-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789."

precedence
   for expression brackets "(" matching ")"
     (same-level "and", "or" associativity left),
     (same-level "<", "<=", "=", ">=", ">", "/=" associativity none),
     (same-level "+","-"  associativity left),
     (same-level "*", "/", "mod"  associativity left),
     (same-level "not"  associativity none),
     (same-level "abs"  associativity none),
     (same-level "**" associativity right)
end
```

D.4 Imports-Exports

```
!! in-package("RU")
!! in-grammar('user)

#||
  File name:   imports-exports.re
    This file encapsulates the import/export-related processing in one place

||#

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
%%   PREPROCESSING
%-------------------------------------------------------------------------
%-------------------------------------------------------------------------


%%  Build-Import-Export-Area -- Considered part of "preprocessing". Called from
%%    semantic-checks to build import/export areas.  It sets up the import area
%%    and export area of the subsystem (input parameter, subsystem).  It
%%    enumerates over all the controllees of the subsystem.  If the controllee
%%    is a primitive object, it adds a new import-obj to the import-area for
%%    each member of the object's INPUT-DATA, if there is not already an
%%    import-obj there with the same id (uses import-symbols to keep track).
%%    Same kind of thing for the exports...

function build-import-export-area (subsystem : subsystem-obj) =

  (enumerate ctrlee over controllees(subsystem) do

     let (obj : component-obj =
                  find-object('component-obj, ctrlee))

        if primitive-obj(obj) then
             % Are there any import items for this primitive object
             %   in subsystem's import area?
             % If not, added object's input-data to import area.
          let (import-set : set(import-obj) =
                { x | (x:import-obj) import-obj(x) &
                                    parent-expr(x) = subsystem &
                                    consumer(x) = name(obj)})
           (if size(import-set) = 0 then
                % No imports yet for this object; add them

             let (input-data : set(import-obj) =
                     get-input-output-variable(obj, "INPUT-DATA"))
             enumerate import over input-data do
               set-attrs(import, 'consumer, name(obj));
               set-attrs(subsystem, 'import-area,
```

```
                    import-area(subsystem) with copy-term(import))
                          % use copy-term so it makes a copy of the object,
                          % not just a pointer to it
          );

            % Are there any export items for this primitive object
            %   in subsystem's export area?
            % If not, add object's output-data to export area.

          let (export-set : set(export-obj) =
               { x | (x:export-obj) export-obj(x) &
                                    parent-expr(x) = subsystem &
                                    producer(x) = name(obj)})
          (if size(export-set) = 0 then
               % No exports yet for this object; add them
           let (output-data : set(export-obj) =
                 get-input-output-variable(obj, "OUTPUT-DATA"))
           enumerate export over output-data do
               set-attrs(export, 'producer, name(obj));
               set-attrs(subsystem, 'export-area,
                    export-area(subsystem) with copy-term(export))
                          % use copy-term so it makes a copy of the object,
                          % not just a pointer to it
          )
    );
       % Now that we've ensured all input-data and output-data for
       %   all primitive object controllees are in import/export area,
       %   must remove any extraneous ones (belonged to primitive objects
       %   which are no longer part of the subsystem)

    let (imports-not-used : set(import-obj) =
          { x | (x:import-obj) import-obj(x) &
                               parent-expr(x) = subsystem &
                               consumer(x) ~in controllees(subsystem)})
    (enumerate import over imports-not-used do
       set-attrs(subsystem, 'import-area, import-area(subsystem) less import)
    );

    let (exports-not-used : set(export-obj) =
          { x | (x:export-obj) export-obj(x) &
                               parent-expr(x) = subsystem &
                               producer(x) ~in controllees(subsystem)})

    (enumerate export over exports-not-used do
       set-attrs(subsystem, 'export-area, export-area(subsystem) less export)
    )

%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
```

```
%%  Determine-Sources-for-Conditionals  --  Considered part of preprocessing.
%%     Called by semantic-checks after the subsystem's import and export areas
%%     are built to associate each identifier (in an if/while condition) with an
%%     import-obj or export-obj in the subsystem's import/export areas.
%%   Each identifier in a conditional must reference an import-obj or export-obj
%%    in its subsystem's import area or export-area (as there is no "get-state"
%%    interface in the OCU model, all identifiers in conditionals must reference
%%    import/export areas, the only data available).  This process is very like
%%    obtaining the source for import-objs...

function determine-sources-for-conditionals ( subsystem : subsystem-obj ) =

  let (identifiers : seq (identifier) =
          [ i | (i:identifier) identifier(i) &
                   least-ancestor-of-class(i, 'subsystem-obj) = subsystem])

  enumerate id over identifiers do
    if undefined?(id-source(id)) then    % do not yet have any source
      let (id-string : string = symbol-to-string(id-name(id)))
      if separator in id-string then  % user has qualified id name
        extract-and-set-id-source(id)
      else
        get-id-source-for-conditional(id)
    else
      format(t, "~%~%There is already a source specified for id ~s",
                     id-name(id));
      format(t, " in conditional expression~%");
      format(t, "                                    in subsystem ~s~%",
               name(least-ancestor-of-class(id, 'subsystem-obj)));
      (if import-obj(id-source(id)) then
         format(t, "    The source is the import item ~s~%",
                   import-name(id-source(id)));
         format(t, "                        consumed by ~s~%",
                   consumer(id-source(id)))
      else
         format(t, "    The source is the export item ~s~%",
                   export-name(id-source(id)));
         format(t, "                        produced by ~s~%",
                   producer(id-source(id)))
      );
      if lisp::y-or-n-p("Do you want to select a different source?") then
         get-id-source-for-conditional(id)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%% Determine-Import-Sources  --  Considered part of "preprocessing".  Called
%%    by semantic-checks if there are no errors to determine which export-obj
%%    will serve as the source of the data to be requested by each import-obj.
%%    If no source currently exists, go figure out which one to use
%%    (via get-source); else, present the currently specified to the user
%%    who may want to (and can) change it.
```

```
%%

function determine-import-sources (subsystem : subsystem-obj) =

 enumerate import over import-area(subsystem) do

   if empty(source(import)) then   % no source currently specified
     get-source(import)

   else

     (if size(source(import)) = 1 then
           % a particular source was selected previously
       let (source-info : source-obj = arb(source(import)))

       format(t, "~%~%There is already a source specified for   ~s ~%",
                  import-name(import));
       format(t, "                which is used by object   ~s ~%",
                  consumer(import));
       format(t, "                             in subsystem   ~s ~%",
                  name(parent-expr(import)));
       format(t, "    The source is:           ~s~%", source-name(source-info));
       format(t, "        produced by object   ~s~%", source-object(source-info));
       format(t, "              in subsystem   ~s~%", source-subsystem(source-info))

      else

        format(t, "~%An arbitrary source is to be used for  ~s~%",
                  import-name(import));
        format(t, "              which is used by object  ~s~%",
                  consumer(import));
        format(t, "                             in subsystem  ~s~%",
                  name(parent-expr(import)))
     );

     if (lisp::y-or-n-p("Do you want to select a different source?")) then
           get-source(import)
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
%%  PREPROCESSING UTILITIES
%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
%%  Get-Source  --  Used by Determine-Import-Sources. First, sets the import's
%%     source to the null set (either it is null to start or the user wants to
%%     wipe out what is already there.  Then find all export-objs within the
%%     same application definition (spec-obj)  which can provide the kind of
%%     data needed by import.  If only one export-obj can be the source, use it.
```

```
%%     Otherwise, present the possible choices to the user who must specify
%%     which one he wants to use.

function get-source(import : import-obj) =

        set-attrs(import, 'source, {});  % wipe out whatever was there

        let (export-seq : seq(export-obj) =
                  [source-export | (source-export:export-obj)
                    export-obj(source-export) &
                    export-category(source-export) = import-category(import) &
                    up-to-root(source-export) = up-to-root(import) &
                    subsystem-obj(parent-expr((source-export)))])
                        % added this last one so it won't find
                        % the import/export-objs which are in the variables
                        % (e.g. THING-OBJ-INPUT)
        if size(export-seq) < 1 then
              %%  Error - should not occur at this time;
              %%  Supposed to catch this error during semantic checks;
              %%  this is called only if there are no errors...
          format(t, "Error: No subsystem provides ~s type of data~%",
                       symbol-to-string(import-category(import)));
          undefined

        elseif size(export-seq) = 1 then

            format(debug-on,
               "There is only one possible source for this info; use it~%");
            set-import-source(import, {export-seq(1)})

        else
              %%  More than one subsystem provides this data-item.
              %%  Prompt the user to select the one to be used...
              %%    and store its name for future reference
              %%  OR if user doesn't care where data comes from, store
              %%    all possible choices...

           let (user-choice : integer = prompt-for-source(import, export-seq))

             (if user-choice <= size(export-seq) then
                    %% User has selected a particular source for this data;
                    %% set source (import) to this selected source only...
                set-import-source(import, {export-seq(user-choice)})
             else
                  %% User has indicated he doesn't care where import data
                  %% comes from; set source(import) to all the possible sources
                set-import-source(import, seq-to-set(export-seq))
             )

%-------------------------------------------------------------------------------
%%  Set-Import-Source  --  loops through all export-objs in set-to-set,
```

```
%%     creating a new source-obj for each element of set-to-set and setting
%%     its attributes (source-subsystem and source-object), based on
%%     info in and about each element of set-to-set.  Called by Get-Source

function set-import-source(import     : import-obj,
                           set-to-set : set(export-obj)) =

  enumerate x over set-to-set do
    let (s : source-obj = make-object('source-obj))
    source-subsystem(s) <- name(parent-expr(x));
    source-object(s)    <- producer(x);
    source-name(s)      <- export-name(x);
    source(import)      <- source(import) with s
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%   Prompt-for-Source  --  given a sequence of export-objs to choose from,
%%      display each possible choice and prompt the user to choose one.
%%      The last choice is always "don't care", i.e., use arbitrary source.
%%      Returns number chosen by the user.

function prompt-for-source (import : import-obj,
                            seq-to-choose-from : seq(export-obj)) : integer =

    format(t, "~%More than one export can provide the data for ~s~%",
                           import-name(import));
    format(t, "                          which is used by object ~s~%",
                           consumer(import));
    format(t, "                                    in subsystem ~s~%",
                           name(parent-expr(import)));
    format(t, "Choose the export item (subsystem and component)~%");
    format(t, "  that you wish to be the source of this data:~%");

    (enumerate index from 1 to size(seq-to-choose-from) do
      format(t, "      ~d> subsystem ~s component ~s name ~s~%",
                 index,
                 symbol-to-string(name(parent-expr(seq-to-choose-from(index)))),
                 symbol-to-string(producer(seq-to-choose-from(index))),
                 symbol-to-string(export-name(seq-to-choose-from(index))))

    );
    format(t, "      ~d> Specific source not required; use arbitrary one~%",
                size(seq-to-choose-from)+1);
    format(t, "Enter the number corresponding to the source you want to use~%");
    read-input()
%-------------------------------------------------------------------------------

%%  The following functions are used when handling identifiers in "if" and
%%  "while" conditions.  As they deal with import and/or export areas, these
%%  functions have been placed in this file to localize any import/export
%%  changes (and there have been a lot of them!).
```

```
%-------------------------------------------------------------------------

%%  Extract-and-Set-Id-Source  --  Separates the consumer/producer from the
%%     id name and uses this info to find the appropriate import or export
%%     object for the id's source.  The separator is a constant which is set
%%     in globals.re

function extract-and-set-id-source ( id : identifier ) =

   let (id-name-string : string = symbol-to-string(id-name(id)))

   let (source-object-name    : string = "",   % consumer/producer name
        source-name           : string = "",   % id name
        position-of-separator : integer = size(id-name-string)+1)

 % Extract the consumer/producer name and id name
   (enumerate index from 1 to size(id-name-string) by 1 do
     if id-name-string(index) = separator then
       position-of-separator <- index
     elseif position-of-separator > size(id-name-string) then
        source-object-name <-
          append(source-object-name, id-name-string(index))
     else
        source-name <- append(source-name, id-name-string(index))
    );

    set-attrs(id, 'id-source, undefined);  % wipe out what was there

    let (subsystem : subsystem-obj =
          least-ancestor-of-class(id, 'subsystem-obj))

    let (import-set : set(object) =
           {import | (import:import-obj) import-obj(import) &
                import in import-area(subsystem) &
                import-name(import) = string-to-symbol(source-name, "RU") &
                consumer(import) = string-to-symbol(source-object-name, "RU")},

       export-set : set(object) =
           {export | (export:export-obj) export-obj(export) &
                export in export-area(subsystem) &
                export-name(export) = string-to-symbol(source-name, "RU") &
                producer(export) = string-to-symbol(source-object-name, "RU")})

    let (possible-choices : seq(object) =
         set-to-seq(import-set union export-set))

    if empty(possible-choices) then
     format(t,
       "There is no possible data in subsystem for conditional identifier ~s ~%",
              id-name(id))
    elseif
```

```
      size(possible-choices) = 1 then  % there is only 1 place to get this info
          set-attrs(id, 'id-source, possible-choices(1));
          set-attrs(id, 'id-name, string-to-symbol(source-name, "RU"))
    else   % The qualification was not precise enough
      format(t, "More than one import/export item meets the qualification.~%");
      format(t, "  Please contact the software engineer/domain engineer.~%")

%-------------------------------------------------------------------------
%%  Get-Id-Source-for-Conditional  --  First, sets the identifier's
%%     source to undefined (either it is null to start or the user wants to
%%     wipe out what is already there).  Then find all import/export-objs within
%%     the subsystem which could be the source of data for this id (i.e., that
%%     have the same name.  If only one can be the source, use it.  Otherwise,
%%     present the possible choices to the user who must specify which one he
%%     wants to use.

function get-id-source-for-conditional ( id : identifier )  =

  set-attrs(id, 'id-source, undefined);  % wipe out what was there

  let (subsystem : subsystem-obj =
          least-ancestor-of-class(id, 'subsystem-obj))

  let (import-set : set(object) =
           {import | (import:import-obj) import-obj(import) &
                import in import-area(subsystem) &
                import-name(import) = id-name(id)},

       export-set : set(object) =
           {export | (export:export-obj) export-obj(export) &
                export in export-area(subsystem) &
                export-name(export) = id-name(id)})

  let (possible-choices : seq(object) =
         set-to-seq(import-set union export-set))

  if empty(possible-choices) then
     format(t,
       "There is no possible data in subsystem for conditional identifier ~s ~%",
             id-name(id))
  else
     if size(possible-choices) = 1 then  % there is only 1 place to get this info
          format(t, "There is only one possible choice for ~s, so select it~%",
             id-name(id));
          set-attrs(id, 'id-source, possible-choices(1))
     else   %  ask the user which import/export obj to use for source
       let (choice : integer =
              prompt-for-conditional-source(id-name(id), possible-choices))
       set-attrs(id, 'id-source, possible-choices(choice))

%-------------------------------------------------------------------------
```

```
%-------------------------------------------------------------------------
%%  Prompt-for-Conditional-Source  --  very similar to Prompt-for-Source
%%     but the printing format and messages are a little different.
%%     Given a sequence of export-objs to choose from,  display each possible
%%     choice and prompt the user to choose one.
%%     Called by get-id-source-for-conditional.

function prompt-for-conditional-source
                        (looking-for        : symbol,
                         seq-to-choose-from : seq(object)) : integer =

  format(t, "There is more than one possibility for data with name ~s~%", looking-for);
  format(t, "  Choose the data you would like to use for evaluating the conditional~%");

  (enumerate index from 1 to size(seq-to-choose-from) do
     if import-obj(seq-to-choose-from(index)) then
       format(t, "      ~d> In import area of subsystem ~s: consumer = ~s: name = ~s~%",
                      index, name(parent-expr(seq-to-choose-from(index))),
                      consumer(seq-to-choose-from(index)),
                      import-name(seq-to-choose-from(index)))
     else
       format(t, "      ~d> In export area of subsystem ~s: producer = ~s: name = ~s~%",
                      index, name(parent-expr(seq-to-choose-from(index))),
                      producer(seq-to-choose-from(index)),
                      export-name(seq-to-choose-from(index)))
  );
  read-input()


%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
%%  ACCESSING IMPORT/EXPORT AREAS  --  Used during behavior simulation (execution)
%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

%%  Get-Import --  returns the value of an external data item.
%%    Given an id-name and consumer, function finds the import-obj associated
%%    with that id.  If the source attribute is defined (we have already
%%    used this id before and know where to get the data), can go directly
%%    to right subsystem's export-area and return the value.  If not,
%%    (we haven't used this id yet), must try to find an export-obj with
%%    the same id in another subsystem.  If there's only one subsystem
%%    with that id in its export-area, that's the one to use: return its
%%    value and set the import-obj source to that subsystem name.  If there
%%    are more than one, prompt the user to specify which subsystem to use as
%%    the source.  Set the source to that subsystem/object (so user doesn't have
%%    to be prompted again) and return the appropriate value.
%%  NOTE:  changes when source was made a set:  if the set is empty (have not
%%    yet tried to access this data), find all possible sources.  Prompt the
%%    user to select one of the possible sources or "arbitrary source".  If
```

```
%%    a particular source was selected, source has only that entry.  If
%%    "arbitrary" was chosen, source contains all the possible choices.  In
%%    either case, select an arbitrary member of the source set.

function get-import (id-name         : symbol,
                     subsystem       : subsystem-obj,
                     consumer-obj    : primitive-obj)  : any-type =

  let (import : import-obj =
          arb({import | (import:import-obj) import-obj(import) &
               import in import-area(subsystem) &
               import-name(import) = id-name
               & consumer(import) = name(consumer-obj)}))

  if undefined?(import) then
       %%  Oops!  This shouldn't happen.  If it does, dm and code for primitive
       %%  object must be checked to ensure compatibility WRT input-data...
     format(t, "A run-time error has occurred.  There is no import-obj for ~s~%",
                     id-name);
     format(t, "  which is used by ~s in subsystem ~s~%", name(consumer-obj),
                     name(subsystem));
     format(t, "Please contact the software engineer~%");
     undefined
  else
     if ~empty(source(import)) then
          %%  We know which subsystem has this info;
          %%    go directly to the right one

     let (s : source-obj =
               arb(source(import)))  % if there is only one source specified,
                                     %   automatically gets the correct one
     let (source-sub : subsystem-obj =
               find-object('subsystem-obj, source-subsystem(s)))
     let (source-export : export-obj =
               arb({source-export | (source-export:export-obj)
                      export-obj(source-export) &
                      source-export in export-area(source-sub) &
                      export-name(source-export) = source-name(s) &
                      producer(source-export) = source-object(s)}))
      (if undefined?(value(source-export)) then
         format(t, "The export item corresponding to ~s which is used by~%",
                   id-name);
         format(t, "  ~s in subsystem ~s has not yet been set.~%",
                   name(consumer-obj), name(subsystem))
      );
      value(source-export)

     else
          %%  We don't yet know which subsystem will provide this info.
          %%  So, there must have been some error in our preprocessing...
       format(t, "A run-time error has occurred.  There is no source for ~s~%",
```

```
                    id-name);
      format(t, "  which is used by ~s in subsystem ~s~%", name(consumer-obj),
                    name(subsystem));
      format(t, "Please contact the software engineer.~%");

      undefined
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Set-Export --  set the value attribute of the export-obj
%%    whose export-name attribute = id-name to val.  This makes val
%%    available to external subsystems.

function set-export (subsystem   : subsystem-obj,
                     source-obj  : primitive-obj,
                     id-name     : symbol,
                     val         : any-type) =

  let (export : export-obj =
         arb({export | (export:export-obj) export-obj(export) &
              export in export-area(subsystem) &
              export-name(export) = id-name &
              producer(export) = name(source-obj)}))

  if undefined?(export) then
       %%  Oops!  This shouldn't happen.  If it does, dm and code for primitive
       %%  object must be checked to ensure compatibility WRT output-data...
     format(t,
       "You have tried to export a value which is not in object's output-data~%")
  else
     set-attrs(export, 'value, val)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Get-Id-Type-For-Conditional  --  If id-source for id has not been specified,
%%     return ERROR to caller (to avoid unusual run-time error; should always
%%     have a source by this time); otherwise, return the data type of the data to
%%     be used as source of id.  Called by eval-expr during semantic checking
%%     of boolean expressions.

function get-id-type-for-conditional ( id : identifier ) : symbol =

  if undefined?(id-source(id)) then
     format(t, "Id ~s has not been associated with an import/export-obj~%",
                  id-name(id));
     'ERROR
  elseif import-obj(id-source(id)) then
     import-type-data(id-source(id))
  else
     export-type-data(id-source(id))
```

```
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Get-Id-Value-for-Conditional  --  Retrieves the current value of id.
%%    If id-source is an import-obj, must used get-import to obtain the value
%%    (since the value isn't stored in an import-obj).  If id-source is an
%%    export-obj, get the value directly for it.  Id-source should already
%%    be defined...

function get-id-value-for-conditional ( id : identifier ) : any-type =

  if undefined?(id-source(id)) then  % something strange is going on
     'ERROR
  elseif import-obj(id-source(id)) then
    get-import(id-name(id), parent-expr(id-source(id)),
                find-object('primitive-obj, consumer(id-source(id))))
  else
     value(id-source(id))
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%%  GENERAL UTILITIES
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

%%  get-input-output-variable -- returns the set of input-data or output-data
%%    associated with that object type.  Each domain object class has two
%%    variables: objectclass-INPUT-DATA and objectclass-OUTPUT-DATA which
%%   define the input-data and output-data associated with domain objects
%%   of that type.  This function constructs the correct variable name
%%   (by concatenating object class (of input parameter, obj), -, and
%%   INPUT-DATA or OUTPUT-DATA (from input parameter, in-or-out)).
%%   It then calls the lisp function, symbol-value, using the constructed
%%   variable name, and returns that value.
%%      NOTE:  Input-data and output-data for each object class MUST follow this
%%             naming convention.

function get-input-output-variable(obj       : primitive-obj,
                                   in-or-out : string) : set(any-type) =

  let( oc : re::binding = instance-of(obj))
    let(var-name : symbol =
       string-to-symbol(concat(symbol-to-string(name(oc)), "-", in-or-out), "ru"))

       symbol-value(var-name)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Get-Source  --  Used by Determine-Import-Sources. First, sets the import's
%%     source to the null set (either it is null to start or the user wants to
%%     wipe out what is already there.  Then find all export-objs within the
```

```
%%      same application definition (spec-obj)  which can provide the kind of
%%      data needed by import.  If only one export-obj can be the source, use it.
%%      Otherwise, present the possible choices to the user who must specify
%%      which one he wants to use.  Called by Determine-Input-Sources.

function get-source(import : import-obj) =

        set-attrs(import, 'source, {});  % wipe out whatever was there

        let (export-seq : seq(export-obj) =
                  [source-export | (source-export:export-obj)
                    export-obj(source-export) &
                    export-category(source-export) = import-category(import) &
                    up-to-root(source-export) = up-to-root(import) &
                    subsystem-obj(parent-expr((source-export)))])
                        % added this last one so it won't find
                        % the import/export-objs which are in the variables
                        % (e.g. THING-OBJ-INPUT)
        if size(export-seq) < 1 then
              %%  Error - should not occur at this time;
              %%  Supposed to catch this error during semantic checks;
              %%  this is called only if there are no errors...
          format(t, "Error: No subsystem provides ~s type of data~%",
                       symbol-to-string(import-category(import)));
          undefined

        elseif size(export-seq) = 1 then

            format(debug-on,
               "There is only one possible source for this info; use it~%");
            set-import-source(import, {export-seq(1)})

        else
              %%  More than one subsystem provides this data-item.
              %%  Prompt the user to select the one to be used...
              %%    and store its name for future reference
              %%  OR if user doesn't care where data comes from, store
              %%    all possible choices...

           let (user-choice : integer = prompt-for-source(import, export-seq))

             (if user-choice <= size(export-seq) then
                    %% User has selected a particular source for this data;
                    %% set source (import) to this selected source only...
                set-import-source(import, {export-seq(user-choice)})
             else
                  %% User has indicated he doesn't care where import data
                  %% comes from; set source(import) to all the possible sources
                set-import-source(import, seq-to-set(export-seq))
             )
%-------------------------------------------------------------------------------
```

D.5 Semantic-Checks

```
!! in-package("RU")
!! in-grammar('user)

%%  File name:  semantic-checks.re


%% The rules that invoke the individual tests on the subsystem assume that the
%% subsystem is the current object (not the spec object).  The main rules
%% (Check-before-executing and reset-fatal-error) can be executed from the
%% spec-obj


%-----------------------------------------------------------------------------
%%  Check-Semantics -- applied on the current node, a spec-obj

rule Check-Semantics (x : object)
   true -->
      Perform-Semantic-Checks(X)
%-----------------------------------------------------------------------------

%-----------------------------------------------------------------------------
%%  Perform-Semantic-Checks  --  enumerates over all the kids of the spec-obj, x,
%%    calling the appropriate check function for the kind of object encountered
%%    (application or subsystem).
%%    There are currently no semantic checks for primitive objects.


function Perform-Semantic-Checks (X : object) =
 %% X is root of abstract syntax tree, spec-obj

  FATAL-ERROR <- false;              % Reset it so only new errors will be flagged
  Semantic-Checks-Performed <- true; % So we know these checks actually were done

  let (components        : seq(symbol) = [],          % used for checking unused components
       application-objs  : seq(application-obj) = [])  % used for checking too many/too few
                                                       %   applications

  (enumerate obj over kids(x) do

    if application-obj(obj) then
      application-objs <- append(application-objs, obj);
      components <- concat(components, [name(obj)],
                           set-to-seq(seq-to-set(application-components(obj))));
      check-application(obj)

    elseif subsystem-obj(obj) then
```

```
      components <- concat(components,
                           set-to-seq(seq-to-set(controllees(obj))));
            % NOTE:  set-to-seq(seq-to-set) is necessary to ensure there is only 1
            %        occurrence of each controllee.  Making controllees a set to start with
            %        did not assure only unique controllee names
      build-import-export-area(obj);  % build framework for im/ex area - "preprocessing"
      determine-sources-for-conditionals(obj);  % more "preprocessing"
      check-subsystem(obj)
  );

%%  Now, all import/export areas in the entire application have been built.
%%  Can check that all imports have a corresponding export
   (enumerate obj over kids(x) do
     if subsystem-obj(obj) then
         Check-for-Exports-Corresponding-to-Imports(obj)
   );

 %%  Do we have one and only one application-obj?
   (if size(application-objs) = 0 then
      Report-Error("There is no application executive in your specification", X)
   else
      if size(application-objs) > 1 then
        Report-Error("There are too many application executives in your specification", X)
   );

 %%  Is a specific primitive object instance used in more than one subsystem?
   (let (dup-seq : seq(symbol) = find-all-dups (components))
    enumerate y over dup-seq do
      if primitive-obj(find-object('component-obj, y)) then
        Report-Error(concat("Object ", symbol-to-string(y),
                    " appears in more than one subsystem"), X)
   );

 %%  Are there any unused components in the spec-obj?
   (let (unused-components : set(component-obj) =
          {z | (z:component-obj) component-obj(z) & name(z) ~in components &
                   up-to-root(z) = X})
    enumerate y over unused-components do
      Report-Warning(concat("Object ", symbol-to-string(name(y)),
                     " is not used in the proposed application"), X)
    );


 %% If no errors so far, determine sources for all imports (part of "preprocessing")
  if ~FATAL-ERROR then
    enumerate component over kids(x) do
      if subsystem-obj(component) then
         determine-import-sources(component)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%%  Check-Application -- ensures application constraints are met, including:
```

```
%%    Check-If-Application-Components-Exist:   (ERROR)
%%         Self-explanatory.
%%    Check-for-Direct-Use-of-Primitives: (ERROR)
%%         Ensures no primitive objects are included in the application directly
%%         (i.e., without an intervening subsystem)
%%    Check-Application-Update-Procedure -- Ensures that operands are part of the
%%         application (i.e., included in application-components) and also includes
%%         the following check:
%%       Check-For-Legal-Call-Statement:   (ERROR)
%%          Ensures that only call statements are included in application-update
%%          (no If or While) and that only implemented subsystem interfaces are
%%          used (now, that's only update!).
%%    Check-For-Dupes-in-Application-Components:   (WARNING)
%%         Checks that each component appears only once.  If not, this may
%%         indicate a specification error (especially a typo), although the
%%         application can be executed.
%%    Check-For-Unused-Components-in-Update:   (WARNING)
%%         Checks for unused components in the update procedure (i.e. components
%%         listed in application-components but not used as operands in the update
%%         procedure).  Again, this could indicate a specification error...

function check-application (application : application-obj) =

   Check-If-Application-Components-Exist (application);
   Check-for-Direct-Use-of-Primitives (application);
   Check-Application-Update-Procedure (application);
   Check-For-Dupes-in-Application-Components (application);
   Check-For-Unused-Subsystems-in-Update (application)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%%  Check-If-Application-Components-Exist  --  Ensures all application-components
%%     exist that should exist. If some do not exist, are they to be initialized
%%     by the application update procedure?  If so, that's OK.
%%     Generates an ERROR...

function check-if-application-components-exist (application : application-obj) =

  enumerate component over application-components(application) do

     if ~object-exists(component) then     % no such subsystem with that name
        Report-Error
           (concat("Object ", symbol-to-string(component), " does not exist"),
              application)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%%  Check-for-Direct-Use-of-Primitives  --  Ensures that no primitive objects are
%%     used directly by the application (application should deal only with subsystems).

function Check-for-Direct-Use-of-Primitives (application : application-obj) =

  enumerate x over application-components(application) do
```

```
     if primitive-obj(find-object('component-obj, x)) then
        Report-Error(concat(symbol-to-string(x),
               " is a primitive; only subsystems can be used in application"),
               application)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%% Check-Application-Update-Procedure  --  Ensures that all statements in
%%     application update procedure are legal.  Includes:

function check-application-update-procedure (application : application-obj) =

  if size(application-update(application)) = 0 then
    Report-Error("No statements in application update procedure", application)
  else
    enumerate stmt over application-update(application) do
      if if-stmt-obj(stmt) then
        Report-Error("If statements are not allowed in application update procedure",
                  application)
      elseif while-stmt-obj(stmt) then
        Report-Error("While statements are not allowed in application update procedure",
                  application)
      else   % have a call statement
        check-if-operand-in-application (application, stmt);
        check-for-legal-call-stmt (stmt)

%-------------------------------------------------------------------------------
%% Check-If-Operand-in-Application  --  Operand of call statement must be included
%%      in application's application-components.

function check-if-operand-in-application (application : application-obj,
                                          stmt        : statement-obj) =

  if operand(stmt) ~in application-components(application) then
    Report-Error (concat ("Object ", symbol-to-string(operand(stmt)),
                          " is not part of the application"), application)
%-------------------------------------------------------------------------------
%%   Check-For-Legal-Call-Stmt  --
%%      Ensures that only subsystem procedural interfaces (update, destroy,
%%      initialize, configure and stabilize (the last four are not yet implemented,
%%      though)) are used in application update procedure.
%%      Generates an ERROR...

function check-for-legal-call-stmt (stmt : statement-obj) =
   if configure-call-obj(stmt) then
       Report-Error ("Configure not yet implemented ", stmt)
   elseif stabilize-call-obj(stmt) then
       Report-Error ("Stabilize not yet implemented ", stmt)
   elseif initialize-call-obj(stmt) then
       Report-Error ("Initialize not yet implemented ", stmt)
   elseif destroy-call-obj(stmt) then
       Report-Error ("Destroy not yet implemented ", stmt)
```

```
   elseif ~update-call-obj(stmt) then
      Report-Error ("Illegal operation in an application update procedure", stmt)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%% Check-for-Dupes-in-Application-Components -- Are any subsystems listed more
%%    than once in application-components? If so, user may have made an error
%%    (most likely a typo). Generates a Warning...

function check-for-dupes-in-application-components (application : application-obj) =

  let (dup-subsystems : seq(symbol) = find-all-dups (application-components(application)))

    if size(dup-subsystems) ~= 0 then  % Have dupes within components of single application
         enumerate dupe over dup-subsystems do
           Report-Warning(concat("Subsystem ", symbol-to-string(dupe),
                                 " appears more than once in the application"), application)

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%% Check-For-Unused-Subsystems-in-Update -- Are any subsystem not used in the
%%    application update procedure? If any are unused, it could indicate an error
%%    by the user (either a typo or perhaps he forgot to include an update
%%    statement). Generates a Warning...

function Check-For-Unused-Subsystems-in-Update (application : application-obj) =

  let (subsystems-unused : set(symbol) = seq-to-set(application-components(application)))

  (enumerate stmt over application-update(application) do
    subsystems-unused <- find-unused-components(subsystems-unused, application, stmt)
  );
  if ~empty(subsystems-unused) then
    enumerate unused over subsystems-unused do
      Report-Warning (concat ("Subsystem ", symbol-to-string(unused),
                              " not used in application update"), application)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%% Check-Subsystem -- ensures subsystem OCU constraints are met.
%%    Check-If-Controllees-Exist:   (ERROR)
%%         Ensures all controllees exist.
%%    Check-Subsystem-Update-Procedure -- includes the following checks
%%       Check-If-Statement:   (ERROR)
%%           Ensures that if-cond is  valid.  Then checks that all statements
%%           in then and else clauses are OK
%%       Check-While-Statement:   (ERROR)
%%           Ensures that if-cond is  valid.  Then checks that all statements
%%           in then and else clauses are OK
%%       Check-If-Operand-in-Subsystem:   (ERROR)
%%           Ensures the operand of call statement is part of the
%%           current subsystem (i.e., it appears in "controllees")
```

```
%%       Check-SetFunction-Stmt:   (ERROR)
%%           Ensures function-name is valid for operand; ensures stmt
%%           coefficients are valid for operand.
%%       Check-SetState-Stmt:   (ERROR)
%%         Ensures state names are valid for operand and the new value
%%         provided for them is of the appropriate type.
%%    Check-For-Dupes-in-Subsystem:   (WARNING)
%%         Checks that each controllee name appears only once.  If not, this may
%%         indicate a specification error (especially a typo), although the application
%%         can be executed.
%%    Check-For-Unused-Components-in-Update:   (WARNING)
%%         Checks for unused components in the update procedure (i.e. components listed
%%         as controllees but not used as operands in the update procedure).  Again,
%%         this could indicate a specification error...

function check-subsystem(subsystem : subsystem-obj) =

   Check-If-Controllees-Exist(subsystem);
   Check-Subsystem-Update-Procedure(subsystem);
   Check-For-Dupes-in-Subsystem(subsystem);
   Check-For-Unused-Components-in-Update(subsystem)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%%  Check-If-Controllees-Exist --
%%    Ensures all controllees exist that should exist.  If some do not exist,
%%    are they to be created by the subsystem?   If so, that's OK.
%%    Generates an ERROR...

function Check-If-Controllees-Exist (subsystem: subsystem-obj) =

  enumerate ctrlee over Controllees(subsystem) do

    if ~object-exists(ctrlee) then  % no such object with the given name
        Report-Error(concat("Object ", symbol-to-string(ctrlee),
              " does not exist"), subsystem)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%%  Check-Subsystem-Update-Procedure  --  Ensures that all statements in
%%     subsystem update procedure are legal.  Includes:

function check-subsystem-update-procedure (subsystem : subsystem-obj) =

  if size(update(subsystem)) = 0 then
    Report-Error("No statements in subsystem update procedure", subsystem)
  else
    enumerate stmt over update(subsystem) do
      check-statement(stmt)

%-------------------------------------------------------------------------------
%%  Check-Statement  --  checks a particular statement.  It's a separate function
%%      so it can be called within if and while statements.
```

```
function check-statement (stmt : statement-obj) =

   if if-stmt-obj(stmt) then
     check-if-statement(stmt)
   elseif while-stmt-obj(stmt) then
     check-while-statement(stmt)
   else   % have a call statement
     check-if-operand-in-subsystem(stmt);
     if setfunction-call-obj(stmt) then
        check-setfunction-stmt(stmt)
     elseif setstate-call-obj(stmt) then
        check-setstate-stmt(stmt)
     elseif ~update-call-obj(stmt) then
        Report-Error("Operation is not yet implemented", stmt)

%-------------------------------------------------------------------------------
%% Check-If-Operand-in-Subsystem  --  Operand of call statement must be included
%%      in subsystem's controllees.

function check-if-operand-in-subsystem (stmt      : statement-obj) =

  let (subsystem : subsystem-obj =
         least-ancestor-of-class(stmt, 'subsystem-obj))

  if operand(stmt) ~in controllees(subsystem) then
    Report-Error (concat ("Object ", symbol-to-string(operand(stmt)),
                          " is not part of the subsystem"), subsystem)
%-------------------------------------------------------------------------------
%% Check-If-Statement  --  Ensure if condition is valid.  Then, ensure all
%%      statements in then and else clauses are OK.

function check-if-statement (stmt : statement-obj) =

   (if get-expression-type(if-cond(stmt)) ~= 'BOOLEAN then
        Report-Error("Invalid condition in if statement ", stmt)
   );
   (enumerate stmt1 over then-stmts(stmt) do
      check-statement(stmt1)
   );
    enumerate stmt2 over else-stmts(stmt) do
      check-statement(stmt2)
%-------------------------------------------------------------------------------
%% Check-While-Statement  --  Ensure while condition is valid.  Then, ensure all
%%      statements in while loop are OK.

function check-while-statement (stmt : statement-obj) =

   (if get-expression-type(while-cond(stmt)) ~= 'BOOLEAN then
        Report-Error("Invalid condition in while statement ", stmt)
   );
```

```
   enumerate stmt1 over while-stmts(stmt) do
      check-statement(stmt1)
%-------------------------------------------------------------------------------
%%  Check-SetFunction-Stmt  --  Ensures that function name specified is valid for
%%      the statement's operand.  Then, ensures that the coefficients specified
%%      (if any) are valid for that operand.

function Check-SetFunction-Stmt (stmt : statement-obj) =

   if object-exists(operand(stmt)) then   %checks meaningful only if operand exists
      check-for-valid-function-name(stmt);
      check-for-valid-coefficients(stmt)


function Check-for-Valid-Function-Name (stmt : statement-obj) =
 %% Thanks to Mary Anne Randour for the essence of this code

   let ( oc : re::binding = instance-of(find-object('primitive-obj, operand(stmt))))
   let (func-name : symbol =
         string-to-symbol (concat(symbol-to-string(name(oc)), "-",
                                  symbol-to-string(function-name(stmt))), "ru"))
   let (valid-function : set(object) =
          { x | (x:object) re::vfunction-op(x) & size(re::formals(x)) = 2 &
                     name(re::ref-to(re::data-type(re::formals(x)(2)))) = name(oc) &
                     name(x) = func-name})
      % to be valid function, must be a function, must have two parameters,
      %   the type of the second parameter must be an object of the same class
      %   as the statement's operand and it must be named func-name (which is
      %   constructed from operand object class and specified function name)

    if empty(valid-function) then
       Report-Error("Invalid function name in setfunction stmt", stmt)


function Check-for-Valid-Coefficients (stmt : statement-obj) =

  let (obj : primitive-obj = find-object('primitive-obj, operand(stmt)))
  let (obj-coefficients : set(name-value-obj) =
         get-computed-attr-value(obj, 'coefficients))
             %NOTE: get-computed-attr-value function is in "execute" file

  enumerate coef over coefficients(stmt) do

    let (c : name-value-obj =
          arb ({c | (c : name-value-obj) name-value-obj(c) &
            c in obj-coefficients & name-value-name(c) = name-value-name(coef)}))
      % to be valid coefficient, there must be a name-value-obj in the operand's
      %    coefficient mapping whose name-value-name = coefficient to be set

    if undefined?(c) then
       Report-Error(concat("Illegal coefficient ",
```

```
                        symbol-to-string(name-value-name(coef))), stmt)
%-------------------------------------------------------------------------
%% Check-SetState-Stmt -- Ensures that state names are valid for operand and
%%    if so, ensures that new value is compatible with the type of data expected
%%    for that state attribute.

function Check-SetState-Stmt (stmt : statement-obj) =

  if object-exists(operand(stmt)) then  %checks meaningful only if operand exists

    let (obj : primitive-obj = find-object('primitive-obj, operand(stmt)))
    let ( oc : re::binding = instance-of(find-object('primitive-obj, operand(stmt))))
#||
    enumerate attr-to-set over state-changes(stmt) do
      let (attribute-name : symbol =
               string-to-symbol (concat(symbol-to-string(name(oc)), "-",
                        symbol-to-string(name-value-name(attr-to-set))), "ru"))
      if ~(ex (attr) (attr in class-attributes(instance-of(obj), true) &
                     name(attr) = attribute-name)) then
         Report-Error("Invalid state name in setstate stmt", stmt)
||#


     enumerate attr-to-set over state-changes(stmt) do
      let (attribute-name : symbol =
               string-to-symbol (concat(symbol-to-string(name(oc)), "-",
                        symbol-to-string(name-value-name(attr-to-set))), "ru"))
      let (attr : re::binding =
            arb({ x | (x:re::binding) re::binding(x) &
                              x in class-attributes(instance-of(obj), true) &
                              name(x) = attribute-name}))
      if undefined?(attr) then
         Report-Error(concat("Invalid state name (",
                            symbol-to-string(name-value-name(attr-to-set)),
                           ") in setstate stmt"), stmt)
      else

  %% Now ensure new value is of the correct type for the attribute specified

      let (legal-type : symbol = get-attribute-type(attr),
           valid-booleans : set(symbol) = {'t, 'f, 'T, 'F})
  format(t, "legal-type = ~s~%", legal-type);
      if legal-type = 'integer and ~integerp(name-value-value(attr-to-set)) then
          Report-Error(concat("Value provided for ",
                              symbol-to-string(name-value-name(attr-to-set)),
                             " is not integer "), attr-to-set)
      elseif legal-type = 'real and ~floatp(name-value-value(attr-to-set)) then
          Report-Error(concat("Value provided for ",
                              symbol-to-string(name-value-name(attr-to-set)),
                             " is not real "), attr-to-set)
      elseif legal-type = 'boolean then
```

```
        (if name-value-value(attr-to-set) ~in valid-booleans then
           Report-Error(concat("Value provided for ",
                               symbol-to-string(name-value-name(attr-to-set)),
                              " is not boolean "), attr-to-set)
        else
           set-attrs(attr-to-set, 'name-value-value,
                     convert-to-boolean(name-value-value(attr-to-set)))
        )
      elseif legal-type = 'string and ~stringp(name-value-value(attr-to-set)) then
        Report-Error(concat("Value provided for ",
                            symbol-to-string(name-value-name(attr-to-set)),
                           " is not string "), attr-to-set)
      elseif legal-type = 'symbol and ~symbolp(name-value-value(attr-to-set)) then
        Report-Error(concat("Value provided for ",
                            symbol-to-string(name-value-name(attr-to-set)),
                           " is not symbol "), attr-to-set)


%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
%%  Check-For-Exports-Corresponding-to-Imports  --  Ensures that for each import-obj
%%     in the subsystem's import-area, an export-obj exists in some subsystem's
%%     export area that corresponds to it (i.e., that can serve as the source of the
%%     external data needed).  Generates an ERROR...
%%   Note:  the import-obj and corresponding export-obj can be part of the same
%%     subsystem.  All subsystems whose export areas are considered for
%%     "correspondence" must be part of the same spec-obj (in case there is more than
%%     one in the object base)


function Check-For-Exports-Corresponding-to-Imports (subsystem : subsystem-obj) =

  enumerate import over import-area(subsystem) do
    let (exports : set(export-obj) =
          { export | (export:export-obj) export-obj(export) &
                  export-category(export) = import-category(import) &
                  up-to-root(export) = up-to-root(import)})

    if empty(exports) then
      Report-Error (concat ("No subsystem produces data of category ",
                            symbol-to-string(import-category(import)), " for object ",
                            symbol-to-string(consumer(import))), subsystem)

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
%%  Check-For-Dupes-in-Subsystem -- Are any controllees listed more than once in the
%%     same subsystem?  If so, user may have made an error (most likely a typo).
%%     Generates a Warning...
```

```
function Check-For-Dupes-in-Subsystem (subsystem : subsystem-obj) =

   let (dup-controllees : seq(symbol) = find-all-dups (controllees(subsystem)))

     if size(dup-controllees) ~= 0 then  % Have dupes within controllees of single subsystem
        enumerate dupe over dup-controllees do
          Report-Warning(concat("Object ", symbol-to-string(dupe),
                                " appears more than once in subsystem"), subsystem)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%% Check-For-Unused-Components-in-Update --  Are any controllees not used in the
%      subsystem update procedure?  If any are unused, it could indicate an error
%%     by the user (either a typo or perhaps he forgot to include an update
%%     statement).  Generates a Warning...


function Check-For-Unused-Components-in-Update (subsystem : subsystem-obj) =

  let (unused-components : set(symbol) = seq-to-set(controllees(subsystem)))

  (enumerate stmt over update(subsystem) do
    unused-components <- find-unused-components(unused-components, subsystem, stmt)
  );
  if ~empty(unused-components) then
    enumerate unused over unused-components do
      Report-Warning (concat ("Component ", symbol-to-string(unused),
                              " not used in subsystem update procedure"), subsystem)
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%% UTILITIES  --  The following functions perform some useful task for the various
%%                semantic checks...
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%% Reset-FATAL-ERROR  --  FATAL-ERROR is global variable in DM.
%%    IF FATAL-ERROR is true, a semantic check has found an error
%%    which must be corrected before the subsystem can be executed.
%%    For test purposes, it is often useful to be able to ignore
%%    these errors by resetting FATAL-ERROR

rule Reset-FATAL-ERROR (x: object)
  FATAL-ERROR -->
       FATAL-ERROR <- false
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%% Report-Error -- used by all semantic checks to display the
%%    error to the user's screen.  Message is the text you wish
%%    displayed and Obj is the object which caused the error
```

```
function Report-Error (Message: string, Obj: object) =

  format(t, "~%ERROR -- ~s ~%", Message);
  format(t, "Object: ~\\pp\\ ~%~%", Obj);
  FATAL-ERROR <- true
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Report-Warning -- used by some semantic checks to display a
%%    warning to the user's screen.  Message is the text you wish
%%    displayed and Obj is the object which caused the error
%%    Note:  A warning indicates something may be wrong with the
%%    specification - user must review to ensure it is written as
%%    intended.

function Report-Warning (Message: string, Obj: object) =

  format(t, "~%Warning -- ~s ~%", Message);
  format(t, "Object: ~\\pp\\ ~%~%",Obj)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
function Object-Exists (Obj-Name : symbol) : boolean =
  ~(empty( Name-Of(Obj-Name)))
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  find-all-dups  --  Given a sequence of any type, returns another sequence
%%     which contains all the duplicates found in the original sequence.
%%     Each duplicate appears only once in the returned sequence -- for example
%%     find-all-dups on [1, 2, 3, 2, 4, 1, 1] returns [1, 2].
%%   Thanks to Dave Zimmerman of Kestrel Institute for this code

function find-all-dups (s: seq(any-type)): seq(any-type) =
   let (var the-dups: seq(any-type) = [])
   s = [.., x, .., y, ..]
   & x = y
   --> x in the-dups;
   the-dups
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
function find-unused-components (unused-components : set(symbol),
                                 obj               : component-obj,
                                 stmt              : statement-obj) : set(symbol) =

  let (components-not-used : set(symbol) = unused-components)

    (if call-obj(stmt) then
       components-not-used <- components-not-used less operand(stmt)
    elseif if-stmt-obj(stmt) then
```

```
      enumerate stmt1 over then-stmts(stmt) do
         components-not-used <- find-unused-components(components-not-used, obj, stmt1);
      enumerate stmt2 over else-stmts(stmt) do
         components-not-used <- find-unused-components(components-not-used, obj, stmt2)
    else
      enumerate stmt1 over while-stmts(stmt) do
         components-not-used <- find-unused-components(components-not-used, obj, stmt1)
    );
    components-not-used
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%% Get-Attribute-Type  --  returns (as a symbol) the data type of an attribute
%%      based on its type in the domain model.
%%      Thanks to Mary Anne Randour for this code!!

function get-attribute-type (attr : re::binding) : symbol =

 let ( type-map : map(symbol, symbol) =
  {| 're::symbol-op   -> 'symbol,
     're::real-op     -> 'real,
     're::integer-op  -> 'integer,
     're::boolean-op  -> 'boolean,
     're::any-type-op -> 'any-type |})

  let (its-type : object = re::range-type(re::data-type(attr)))

  if re::class(its-type) = 're::binding-ref then
    if defined?(re::bindingname(its-type)) and-then
         re::bindingname(its-type) = 'string then
      'string
    else
      'object
  else
    type-map(re::class(its-type))
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%% Convert-to-Boolean  --  transforms a symbol to boolean

function convert-to-boolean ( symbol-to-convert : symbol) : boolean =

  if symbol-to-convert = 't or symbol-to-convert = 'T then
    true
  elseif symbol-to-convert = 'f or symbol-to-convert = 'F then
    false
%-------------------------------------------------------------------------------
```

D.6 Execute

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  execute.re

%-------------------------------------------------------------------------------
%%  Do-Execute -- Used to simulate execution of the entire application.
%%     Current node is the spec-obj parsed in by user.  Semantic checks
%%     must have already been performed with no errors.

rule Do-Execute (X : object)
  ~fatal-error & semantic-checks-performed  --> Find-and-Execute-Application(X)
%-------------------------------------------------------------------------------


%-------------------------------------------------------------------------------
%%   Find-and-Execute-Application  --  Finds the application-obj within the current
%%      application (spec-obj) and calls Execute-Application to execute it.
%%      NOTE:  There is only one application-obj per application -
%%      semantic checks insure that.

function Find-and-Execute-Application (x : object) =

  semantic-checks-performed <- false;  % reset flag to force semantic-checks again

  let (application : application-obj =
        arb({a | (a:application-obj) application-obj(a) & parent-expr(a) = x}))

  Execute-Application(application)

%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Execute-Application -- Simulates execution of an application
%%    (given by the input parmeter, application) by enumerating
%%    over the statements in the application update procedure.
%%    As you can see, this is virtually identical to Execute-Subsystem.
%%    However, we are currently using only the most simple application
%%    executive.  In the future, the execution of applications and subsystems
%%    may be vastly different.

function Execute-Application ( application : object) =

    enumerate stmt over application-update(application) do
      execute-statement(application, stmt)
%-------------------------------------------------------------------------------
```

```
%-------------------------------------------------------------------------------
%%  Execute-Subsystem -- Simulates execution of a subsystem
%%    (given by the input parmeter, subsystem) by enumerating
%%    over the statements in the subsystem update procedure.

function Execute-Subsystem (subsystem : object) =

    enumerate stmt over update(subsystem) do
      execute-statement(subsystem, stmt)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Execute-Statement --  Given a statement from a subsystem update
%%    procedure, calls the appropriate function to execute the
%%    statement.

function Execute-Statement (component  : object,
                            stmt       : statement-obj) =

    if call-obj(stmt) then Do-Call-Stmt(component, stmt)
    elseif if-stmt-obj(stmt) then Do-If-Stmt(component, stmt)
    elseif while-stmt-obj(stmt) then Do-While-Stmt(component, stmt)
    else
      format(t, "RUN-TIME ERROR: trying to execute ");
      format(t, "an invalid statement: ~\\pp\\~%", component)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Do-Call-Stmt -- Executes a Call statement.  Finds the object
%%    referenced as the operand of the call statement.  If that object
%%    is a primitive-obj, call the correct function. If the operand is
%%    another subsystem, recursively call Execute-Subsystem.
%%      Call-stmts include everything but if and while statements...

function Do-Call-Stmt (control: object, stmt : statement-obj) =

 let (obj : object = find-object('component-obj, operand(stmt)))

 if defined?(obj) then
   if primitive-obj(obj) then
     if update-call-obj(stmt) then
           % call the current update function name
           % which is stored in object's update-function attribute
       lisp::funcall(get-computed-attr-value(obj, 'UPDATE-FUNCTION),
                     control, obj)

     elseif SetFunction-call-obj(stmt) then
         SetFunction(operand(stmt), function-name(stmt), coefficients(stmt))

     elseif SetState-call-obj(stmt) then
         SetState(operand(stmt), state-changes(stmt))
```

```
      else
         format(t, "Erroneous call statement for a primitive object ~\\pp\\~%", stmt)

    else   %  operand is a subsystem
      execute-subsystem(obj)

  else  % The object referenced by the operand does not exist
      format(t, "RUN-TIME ERROR: trying to execute a statement ");
      format(t, "with a non-existent operand ~\\pp\\~%", stmt)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Do-If-Stmt --  Executes an If statement.  If if-cond evaluates to
%%     true, call Execute-Statement for each of the statements in
%%     then-stmts.  If if-cond is false, call Execute-Statement for
%%     each statement in else-stmts.

function Do-If-Stmt (control: object, stmt : statement-obj) =

  if evaluate-boolean-expression(if-cond(stmt)) then
    enumerate then-stmt over then-stmts(stmt) do
      execute-statement(control, then-stmt)
  else
    enumerate else-stmt over else-stmts(stmt) do
      execute-statement(control, else-stmt)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Do-While-Stmt -- As long as the while-cond evaluates to true,
%%     call Execute-Statement for each statement in while-stmts.

function Do-While-Stmt (control: object, stmt : statement-obj) =

  while evaluate-boolean-expression(while-cond(stmt)) do
    enumerate while-stmt over while-stmts(stmt) do
      execute-statement(control, while-stmt)
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  SetFunction  --  Only valid for primitive objects.  Sets the object's update
%%      function name and any coefficients that are also provided.  Coefficients
%%      to be set must already exist.

function SetFunction (operand-name        : symbol,
                      function-name       : symbol,
                      coefficients-to-set : set(name-value-obj)) =

  format(debug-on, "Calling SetFunction on primitive object  ~s~%", operand-name);

  let (obj : primitive-obj = find-object('primitive-obj, operand-name))
```

```
  let (oc : re::binding = instance-of(obj))
  let (new-function-name : symbol =
            string-to-symbol(concat(symbol-to-string(name(oc)), "-",
                                    symbol-to-string(function-name)), "ru"))
%%  The above "let" statements are needed to complete the full function name.
%%  The object class must be appended to the function name provided by the user.

  set-computed-attr (obj, 'update-function, new-function-name);

%%  Now set the coefficients...

  let (obj-coefficients : set(name-value-obj) =
           get-computed-attr-value(obj, 'coefficients))

  enumerate coef over coefficients-to-set do

    let (c : name-value-obj =
          arb ({c | (c : name-value-obj) name-value-obj(c) &
            c in obj-coefficients & name-value-name(c) = name-value-name(coef)}))

    if undefined?(c) then
       format(t, "RUN-TIME ERROR: Coefficient ~s does not exist~%",
                    name-value-name(coef))
    else
      obj-coefficients <- obj-coefficients less c;
      obj-coefficients <- obj-coefficients with coef;
      set-computed-attr (obj, 'coefficients, obj-coefficients)

%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  SetState  --  Only valid for primitive objects.  For each item in
%%     state-changes-to-make (statement's state-changes), set the
%%     appropriate attribute of the operand to the given value.

function SetState (operand-name              : symbol,
                   state-changes-to-make     : set(name-value-obj)) =

  let (obj : primitive-obj = find-object('primitive-obj, operand-name))

  format(debug-on, "~%Calling SetState on primitive object ~\\pp\\~%", obj);

  enumerate state-change over state-changes-to-make do
     set-computed-attr (obj, name-value-name(state-change),
                             name-value-value(state-change))
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%%   Utilities
%-------------------------------------------------------------------------------
```

```
%-------------------------------------------------------------------------------

%%  The following functions are "utilities" used to retrieve data
%%  from an attribute whose name must be constructed and to store
%%  data to such an attribute.
%-------------------------------------------------------------------------------
%%  Get-Computed-Attr-Value  --  appends partial-attr-name after
%%     obj's object-class name (a symbol, which is obtained by using
%%     REFINE's instance-of function) to determine attribute name.
%%     Use the REFINE function, retrieve-attribute, to get the current
%%     value stored in that attribute and return it to the calling function.

function get-computed-attr-value (obj               : primitive-obj,
                                  partial-attr-name : symbol ) : any-type  =

  let ( oc : re::binding = instance-of(obj))
    let (var-name : symbol =
         string-to-symbol (concat(symbol-to-string(name(oc)), "-",
                                  symbol-to-string(partial-attr-name)), "ru"))
      retrieve-attribute(obj, find-attribute(var-name))

%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Set-Computed-Attr  --  appends partial-attr-name after obj's
%%     object-class name (a symbol, which is obtained by using
%%     REFINE's instance-of function) to determine the attribute name
%%     (var-name).  Use REFINE's store-attribute function set the
%%     current value of attribute referenced by var-name to new-value.

function set-computed-attr (obj               : primitive-obj,
                            partial-attr-name : symbol,
                            new-value : any-type) =


  let( oc : re::binding = instance-of(obj))
    let(var-name : symbol =
       string-to-symbol (concat(symbol-to-string(name(oc)), "-",
          symbol-to-string(partial-attr-name)), "ru"))
     store-attribute(obj, find-attribute(var-name), new-value)

%-------------------------------------------------------------------------------
%%  Get-Coefficient-Value  --  Used by primitive object update functions to
%%     obtain the current value of the coefficient given by coefficient-name.

function get-coefficient-value (obj             : primitive-obj,
                                coefficient-name : symbol) : any-type =

  let (obj-coefficients : set(name-value-obj) =
           get-computed-attr-value(obj, 'coefficients))
```

```
    let (c : name-value-obj =
          arb ({c | (c : name-value-obj) name-value-obj(c) &
            c in obj-coefficients & name-value-name(c) = coefficient-name}))

   name-value-value(c)
%-------------------------------------------------------------------------------
```

D.7 Eval-Expr

```
!! in-package("RU")
!! in-grammar('user)

%%  file:  eval-expr.re
%%    NOTE:  This code was taken almost verbatim from our final project
%%           in CSCE 663.  This is another example of code reuse...


%-------------------------------------------------------------------------------
%%  Evaluate-Boolean-Expression  --  returns the value of the
%     given boolean expression.  If it's a boolean literal, return
%     its value; if an identifier, retrieve its value (identifiers
%     are linked to an import-obj or export-obj as determined by the
%     application specialist); otherwise, evaluate argument1 and
%     argument2 separately and perform the appropriate operation on
%     them.

function Evaluate-Boolean-Expression (X: Object) : Boolean =

 format(debug-on, "in evaluate boolean  exp, object = ~\\pp\\ ~%", x);
  if True-Literal(X) then
    true

  elseif False-Literal(X) then
    false

  elseif Identifier(X) then
    Get-Id-Value-for-Conditional(X)   % function located in imports-exports

  elseif Equal-Exp(X) then
    let (Exp-Type : symbol = 'INTEGER)
     (if Identifier(Argument1(X)) then
       Exp-Type <- Get-Id-Type-for-Conditional(Argument1(X))
     else % must be an object
       Exp-Type <- Get-Expression-Type(Argument1(X))
     );
     (if Exp-Type = 'INTEGER then
       Evaluate-Integer-Expression(Argument1(X)) =
           Evaluate-Integer-Expression(Argument2(X))
      elseif Exp-Type = 'BOOLEAN then
```

```
     Evaluate-Boolean-Expression(Argument1(X)) =
        Evaluate-Boolean-Expression(Argument2(X))
    elseif Exp-Type = 'STRING then
     Evaluate-String-Expression(Argument1(X)) =
        Evaluate-String-Expression(Argument2(X))
    else % Exp-Type must be real
     Evaluate-Real-Expression(Argument1(X)) =
        Evaluate-Real-Expression(Argument2(X))
   )
elseif LT-Exp(X) then
  let (Exp-Type : symbol = 'INTEGER)
   (if Identifier(Argument1(X)) then
     Exp-Type <- Get-Id-Type-for-Conditional(Argument1(X))
   else % must be an object
     Exp-Type <- Get-Expression-Type(Argument1(X))
   );
   (if Exp-Type = 'INTEGER then
     Evaluate-Integer-Expression(Argument1(X)) <
          Evaluate-Integer-Expression(Argument2(X))
    elseif Exp-Type = 'STRING then
     Evaluate-String-Expression(Argument1(X)) <
          Evaluate-String-Expression(Argument2(X))
    else % Exp-Type must be real
     Evaluate-Real-Expression(Argument1(X)) <
         Evaluate-Real-Expression(Argument2(X))
   )

elseif LTE-Exp(X) then
  let (Exp-Type : symbol = 'INTEGER)
   (if Identifier(Argument1(X)) then
     Exp-Type <- Get-Id-Type-for-Conditional(Argument1(X))
   else % must be an object
     Exp-Type <- Get-Expression-Type(Argument1(X))
   );
   (if Exp-Type = 'INTEGER then
     Evaluate-Integer-Expression(Argument1(X)) <=
          Evaluate-Integer-Expression(Argument2(X))
    elseif Exp-Type = 'STRING then
     Evaluate-String-Expression(Argument1(X)) <
          Evaluate-String-Expression(Argument2(X))
    else % Exp-Type must be real
     Evaluate-Real-Expression(Argument1(X)) <=
         Evaluate-Real-Expression(Argument2(X))
   )
elseif GT-Exp(X) then
  let (Exp-Type : symbol = 'INTEGER)
   (if Identifier(Argument1(X)) then
     Exp-Type <- Get-Id-Type-for-Conditional(Argument1(X))
   else % must be an object
     Exp-Type <- Get-Expression-Type(Argument1(X))
   );
```

```
    (if Exp-Type = 'INTEGER then
      Evaluate-Integer-Expression(Argument1(X)) >
          Evaluate-Integer-Expression(Argument2(X))
     elseif Exp-Type = 'STRING then
      Evaluate-String-Expression(Argument1(X)) <
           Evaluate-String-Expression(Argument2(X))
     else % Exp-Type must be real
      Evaluate-Real-Expression(Argument1(X)) >
          Evaluate-Real-Expression(Argument2(X))
    )

  elseif GTE-Exp(X) then
    let (Exp-Type : symbol = 'INTEGER)
     (if Identifier(Argument1(X)) then
       Exp-Type <- Get-Id-Type-for-Conditional(Argument1(X))
     else % must be an object
       Exp-Type <- Get-Expression-Type(Argument1(X))
     );
     (if Exp-Type = 'INTEGER then
       Evaluate-Integer-Expression(Argument1(X)) >=
           Evaluate-Integer-Expression(Argument2(X))
      elseif Exp-Type = 'STRING then
       Evaluate-String-Expression(Argument1(X)) <
            Evaluate-String-Expression(Argument2(X))
      else % Exp-Type must be real
       Evaluate-Real-Expression(Argument1(X)) >=
           Evaluate-Real-Expression(Argument2(X))
     )

  elseif And-Exp(X) then
    Evaluate-Boolean-Expression(Argument1(X)) &
        Evaluate-Boolean-Expression(Argument2(X))

  elseif  Or-Exp(X) then
    Evaluate-Boolean-Expression(Argument1(X)) or
       Evaluate-Boolean-Expression(Argument2(X))

  elseif Not-Exp(X) then
   not (Evaluate-Boolean-Expression(Argument(X)))
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
%% Evaluate-Integer-Expression  --  returns the value of the
%     given integer expression.  If it's a literal, return
%     its value; if an identifier, retrieve its value (identifiers
%     are linked to an import-obj or export-obj as determined by the
%     application specialist); otherwise, evaluate argument1 and
%     argument2 separately and perform the appropriate operation on
%     them.
```

```
function Evaluate-Integer-Expression (X : Expression) : integer =

 format(debug-on, "in evaluate integer exp, object = ~\\pp\\ ~%", x);

  if Integer-Literal(X) then
    Int-Value(X)

  elseif Identifier(X) then
    Get-Id-Value-for-Conditional(X)    % function located in imports-exports

  elseif add-exp(X) then
    (Evaluate-Integer-Expression(Argument1(X)) +
      Evaluate-Integer-Expression(Argument2(X)))

  elseif Subtract-Exp(X)  then
    Evaluate-Integer-Expression(Argument1(X)) -
      Evaluate-Integer-Expression(Argument2(X))

  elseif Multiply-Exp(X) then
    Evaluate-Integer-Expression(Argument1(X)) *
      Evaluate-Integer-Expression(Argument2(X))

  elseif Divide-Exp(X) then
    Evaluate-Integer-Expression(Argument1(X)) div
      Evaluate-Integer-Expression(Argument2(X))

  elseif Mod-Exp(X) then
    Evaluate-Integer-Expression(Argument1(X)) mod
      Evaluate-Integer-Expression(Argument2(X))

  elseif Abs-Exp(X) then
     let (Exp-Value : integer = 0)
         Exp-Value <- Evaluate-Integer-Expression (Argument1(X));
        if Exp-Value < 0 then
          0 - Exp-Value
        else
          Exp-Value

  elseif Negate-Exp(X) then
     let (Exp-Value : integer = 0)
         Exp-Value <- Evaluate-Integer-Expression (Argument(X));
         0 - Exp-Value

  elseif Positive-Exp(X) then
     let (Exp-Value : integer = 0)
         Exp-Value <- Evaluate-Integer-Expression (Argument(X));
         Exp-Value

  % Refine does not handle exponentiation
  elseif Exponential-Exp(X) then
   let (Power : integer = 1,
```

```
        Base  : integer = Evaluate-Integer-Expression(Argument1(X)),
        Expon : integer = Evaluate-Integer-Expression(Argument2(X)))

    (enumerate index from 1 to Expon do
       Power <- Power * Base );
    Power

  else
   format(debug-on,
      "Trying to evaluate a non-integer function as an integer ~%");
   format(debug-on, "Object = ~\\pp\\ ~%", X);
   0
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
%% Evaluate-String-Expression  --  returns the value of the
%     given string expression.  If it's a literal, return
%     its value; if an identifier, retrieve its value (identifiers
%     are linked to an import-obj or export-obj as determined by the
%     application specialist).  This implementation currently allows
%     no operations on strings within expressions.

function Evaluate-String-Expression (X: Object) : String =

 format(debug-on, "in evaluate string exp, object = ~\\pp\\ ~%", x);

 if String-Literal(X) then
   String-Value(X)

 else  % X is an Identifier
    Get-Id-Value-for-Conditional(X)    % function located in imports-exports
%-------------------------------------------------------------------------

%-------------------------------------------------------------------------
%% Evaluate-Real-Expression  --  returns the value of the
%     given real expression.  If it's al iteral, return
%     its value; if an identifier, retrieve its value (identifiers
%     are linked to an import-obj or export-obj as determined by the
%     application specialist); otherwise, evaluate argument1 and
%     argument2 separately and perform the appropriate operation on
%     them.

function Evaluate-Real-Expression (X: Object) : Real =

 format(debug-on, "in evaluate real exp, object = ~\\pp\\ ~%", x);

 if Real-Literal(X) then
   Real-Value(X)

  elseif Identifier(X) then
    Get-Id-Value-for-Conditional(X)    % function located in imports-exports
```

```
  elseif add-exp(X) then
    (Evaluate-Real-Expression(Argument1(X)) +
       Evaluate-Real-Expression(Argument2(X)))

  elseif Subtract-Exp(X)  then
    Evaluate-Real-Expression(Argument1(X)) -
      Evaluate-Real-Expression(Argument2(X))

  elseif Multiply-Exp(X) then
    Evaluate-Real-Expression(Argument1(X)) *
      Evaluate-Real-Expression(Argument2(X))

  elseif Divide-Exp(X) then
    Evaluate-Real-Expression(Argument1(X)) /
      Evaluate-Real-Expression(Argument2(X))

  elseif Negate-Exp(X) then
    let (Exp-Value : real = 0.0)
         Exp-Value <- Evaluate-Real-Expression (Argument(X));
         0.0 - Exp-Value

  elseif Exponential-Exp(X) then
   let (Power : real = 1.0,
        Base  : real = Evaluate-Real-Expression(Argument1(X)),
        Expon : integer = Evaluate-Integer-Expression(Argument2(X)))

    (enumerate index from 1 to Expon do
       Power <- Power * Base );
    Power

  elseif Abs-Exp(X) then
     let (Exp-Value : real = 0.0)
          Exp-Value <- Evaluate-Real-Expression (Argument1(X));
        if Exp-Value < 0.0 then
           0.0 - Exp-Value
        else
           Exp-Value

  else
   format(debug-on, "Trying to evaluate a non-real function as a real ~%");
   format(debug-on, "Object = ~\\pp\\ ~%", X);
   0.0

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%   Utilities
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

function Report-Type-Mismatch-Error(X:object, left,right : symbol, Message: string) =
```

```
  format(t, "Error -- Type Mismatch in expression -- ~s ~%", Message);
  format(t, "Object: ~\\pp\\ ~%", X);
  format(t, "LHS type is ~s ~%", symbol-to-string(left));
  if right ~= 'nil then
    format(t, "RHS type is ~s ~%~%",  symbol-to-string(right))

%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%%  Get-Expression-Type  --  Returns the type of an expression (as a symbol).
%%      If it finds a mismatch, it reports an error to the calling function.

function Get-Expression-Type (X : object) : symbol =


  if Integer-Literal(X) then
    'INTEGER
  elseif Real-Literal(X) then
    'REAL
  elseif Boolean-Literal(X) then
    'BOOLEAN
  elseif String-Literal(X) then
    'STRING
  elseif Identifier(X) then
     Get-Id-Type-for-Conditional(X)  % returns type of data to which
                                     % id refers or ERROR if no id-source
                                     % is specified

 % for +, -, *, and /, both left and right sides must be the same type
  elseif add-exp(X) or subtract-exp(X) or multiply-exp(X) or divide-exp(X) then
   let (left-type : symbol = Get-Expression-Type(argument1(x)),
        rt-type : symbol =  Get-Expression-Type(argument2(X))  )
     if (left-type = rt-type) and (left-type ~= 'ERROR and rt-type ~= 'ERROR) then
       left-type
     else
      Report-Type-Mismatch-Error(X,left-type, rt-type, "Types must match");
      'ERROR % return error

  % for mod, both must be integers
  elseif mod-exp(X) then
   let (left-type : symbol = Get-Expression-Type(argument1(X)),
        rt-type : symbol =  Get-Expression-Type(argument2(X))  )
     if left-type = 'INTEGER and rt-type = 'INTEGER and
                (left-type ~= 'ERROR and rt-type ~= 'ERROR) then
       left-type
     else
      Report-Type-Mismatch-Error(X,left-type, rt-type, "Both sides must be integers");
      'ERROR % return error

  % for **, left side can be integer or real, right side must be integer.
```

```
elseif exponential-exp(X) then
 let (left-type : symbol = Get-Expression-Type(argument1(X)),
      rt-type : symbol =  Get-Expression-Type(argument2(X))  )
   if (left-type = 'INTEGER or left-type = 'REAL) and rt-type = 'INTEGER
         and (left-type ~= 'ERROR and rt-type ~= 'ERROR) then
     left-type
   else
    Report-Type-Mismatch-Error(X,left-type, rt-type,
        "Left side must be integer or real and right side must be an integer");
   'ERROR % return error

% for abs, argument cannot be boolean
elseif Abs-Exp(X) then
  let (Exp-Type : symbol = Get-Expression-Type(Argument(X)) )
    if Exp-Type ~= 'BOOLEAN and Exp-Type ~= 'STRING then
      Exp-Type
    else
      Report-Type-Mismatch-Error(X, Exp-Type, 'nil,
           "Type must not be boolean or string");
     'ERROR % return error

% for unary minus, argument cannot be boolean
elseif Negate-Exp(X) then
  let (Exp-Type : symbol = Get-Expression-Type(Argument(X)) )
    if Exp-Type ~= 'BOOLEAN and Exp-Type ~= 'STRING then
      Exp-Type
    else
      Report-Type-Mismatch-Error(X, Exp-Type, 'nil,
           "Type must not be boolean or string");
     'ERROR % return error

% for unary plus, argument cannot be boolean
elseif Positive-Exp(X) then
  let (Exp-Type : symbol = Get-Expression-Type(Argument(X)) )
    if Exp-Type ~= 'BOOLEAN and Exp-Type ~= 'STRING then
      Exp-Type
    else
      Report-Type-Mismatch-Error(X, Exp-Type, 'nil,
           "Type must not be boolean or string");
     'ERROR % return error


% all relation operators, both arguments must be the same type
elseif Equal-Exp(X) or Not-Equal-Exp(X) or LT-Exp(X) or LTE-Exp(X) or
          GT-Exp(X) or GTE-Exp(X) then
 let (left-type : symbol = Get-Expression-Type(argument1(X)),
      rt-type : symbol =  Get-Expression-Type(argument2(X))  )
   if left-type = rt-type  and (left-type ~= 'ERROR and rt-type ~= 'ERROR) then
     'BOOLEAN
   else
    Report-Type-Mismatch-Error(X,left-type, rt-type, "Types must match");
```

```
   'ERROR % return error

% for and, or both arguments must be booleans
elseif And-Exp(X) or Or-Exp(X) then
 let (left-type : symbol = Get-Expression-Type(argument1(X)),
      rt-type : symbol =  Get-Expression-Type(argument2(X))  )
   if left-type = 'BOOLEAN and rt-type = 'BOOLEAN
            and (left-type ~= 'ERROR and rt-type ~= 'ERROR) then
     'BOOLEAN
   else
    Report-Type-Mismatch-Error(X,left-type, rt-type, "Both sides must be booleans");
   'ERROR % return error

% for not, argument must be boolean
elseif Not-Exp(X) then
  let (Exp-Type : symbol = Get-Expression-Type(Argument(X)) )
    if Exp-Type = 'BOOLEAN then
      Exp-Type
    else
      Report-Type-Mismatch-Error(X, Exp-Type, 'nil, "Type must be boolean");
     'ERROR % return error

else
  format(debug-on,
      "You shouldn't be calling this procedure with this object!!~%");
  format(debug-on, "Object = ~\\pp\\ ~%", x)
```

Appendix E. *Technology Base for the Logic Circuit Domain*

This appendix contains the REFINE code for the logic circuit technology based used in the logic circuit domain used to validate Architect. Each section defines a single primitive object within the domain. Please note the conformance to the primitive object template described in Appendix A.

E.1 And-Gate

```
!! in-package("RU")
!! in-grammar('user)


%% File name:  and-gate.re

 var AND-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var AND-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var AND-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var AND-GATE-OBJ-COEFFICIENTS : map(AND-GATE-OBJ, set(name-value-obj))
  computed-using
    AND-GATE-OBJ-COEFFICIENTS(x) = {}


 var AND-GATE-OBJ-UPDATE-FUNCTION : map(AND-GATE-OBJ, symbol)
   computed-using
     AND-GATE-OBJ-UPDATE-FUNCTION(x) = 'AND-GATE-OBJ-UPDATE1

% Other Attributes:
 var AND-GATE-OBJ-DELAY : map(AND-GATE-OBJ, integer)
   computed-using
```

```
   AND-GATE-OBJ-DELAY(x) = 0

 var AND-GATE-OBJ-MANUFACTURER :  map(AND-GATE-OBJ, string)
   computed-using
    AND-GATE-OBJ-MANUFACTURER(x) = " "

 var AND-GATE-OBJ-MIL-SPEC? :  map(AND-GATE-OBJ, boolean)
   computed-using
    AND-GATE-OBJ-MIL-SPEC?(x) = nil

 var AND-GATE-OBJ-POWER-LEVEL :  map(AND-GATE-OBJ, real)
   computed-using
    AND-GATE-OBJ-POWER-LEVEL(x) = 0.0

 form Make-AND-GATE-Names-Unique
  unique-names-class('AND-GATE-OBJ, true)


%-----------------------------------------------------------------------
function AND-GATE-OBJ-UPDATE1 (subsystem : subsystem-obj,
                               and-gate  : AND-GATE-OBJ) =

 format(debug-on, "AND-GATE-OBJ-UPDATE on ~s~%", name(and-gate));

 let (in1 : boolean = get-import('in1, subsystem, and-gate),
      in2 : boolean = get-import('in2, subsystem, and-gate))

 set-export(subsystem, and-gate, 'out1, in1 & in2)
%-----------------------------------------------------------------------

function AND-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                  and-gate  : AND-GATE-OBJ) =

 format(t, "AND-GATE-OBJ-NEW-UPDATE on ~s~%", name(and-gate))
```

E.2 Or-Gate

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  or-gate.re

 var OR-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var OR-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
```

```
                    'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                    'import-name, 'in2,
                    'import-category, 'signal,
                    'import-type-data, 'boolean)}

 var OR-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                    'export-name, 'out1,
                    'export-category, 'signal,
                    'export-type-data, 'boolean)}

 var OR-GATE-OBJ-COEFFICIENTS : map(OR-GATE-OBJ, set(name-value-obj))
  computed-using
    OR-GATE-OBJ-COEFFICIENTS(x) = {}


 var OR-GATE-OBJ-UPDATE-FUNCTION : map(OR-GATE-OBJ, symbol)
   computed-using
     OR-GATE-OBJ-UPDATE-FUNCTION(x) = 'OR-GATE-OBJ-UPDATE1

% Other Attributes:
 var OR-GATE-OBJ-DELAY : map(OR-GATE-OBJ, integer)
   computed-using
    OR-GATE-OBJ-DELAY(x) = 0

 var OR-GATE-OBJ-MANUFACTURER :  map(OR-GATE-OBJ, string)
   computed-using
    OR-GATE-OBJ-MANUFACTURER(x) = " "

 var OR-GATE-OBJ-MIL-SPEC? :  map(OR-GATE-OBJ, boolean)
   computed-using
    OR-GATE-OBJ-MIL-SPEC?(x) = nil

 var OR-GATE-OBJ-POWER-LEVEL :  map(OR-GATE-OBJ, real)
   computed-using
    OR-GATE-OBJ-POWER-LEVEL(x) = 0.0

 form Make-OR-GATE-Names-Unique
  unique-names-class('OR-GATE-OBJ, true)
%-----------------------------------------------------------------
function OR-GATE-OBJ-UPDATE1 (subsystem : subsystem-obj,
                              or-gate   : OR-GATE-OBJ) =

 format(debug-on, "OR-GATE-OBJ-UPDATE on ~s~%", name(or-gate));

 let (in1 : boolean = get-import('in1, subsystem, or-gate),
      in2 : boolean = get-import('in2, subsystem, or-gate))

 set-export(subsystem, or-gate, 'out1, (in1 or in2))
```

```
%-------------------------------------------------------------------
function OR-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                 or-gate   : OR-GATE-OBJ) =

 format(t, "OR-GATE-OBJ-NEW-UPDATE on ~s~%", name(or-gate))
```

E.3 Nand-Gate

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  nand-gate.re

 var NAND-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var NAND-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

        set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var NAND-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var NAND-GATE-OBJ-COEFFICIENTS : map(NAND-GATE-OBJ, set(name-value-obj))
  computed-using
    NAND-GATE-OBJ-COEFFICIENTS(x) = {}


 var NAND-GATE-OBJ-UPDATE-FUNCTION : map(NAND-GATE-OBJ, symbol)
   computed-using
     NAND-GATE-OBJ-UPDATE-FUNCTION(x) = 'NAND-GATE-OBJ-UPDATE1

% Other Attributes:
 var NAND-GATE-OBJ-DELAY : map(NAND-GATE-OBJ, integer)
   computed-using
    NAND-GATE-OBJ-DELAY(x) = 0

 var NAND-GATE-OBJ-MANUFACTURER :  map(NAND-GATE-OBJ, string)
   computed-using
```

```
   NAND-GATE-OBJ-MANUFACTURER(x) = " "

 var NAND-GATE-OBJ-MIL-SPEC? :  map(NAND-GATE-OBJ, boolean)
   computed-using
    NAND-GATE-OBJ-MIL-SPEC?(x) = nil

 var NAND-GATE-OBJ-POWER-LEVEL :  map(NAND-GATE-OBJ, real)
   computed-using
    NAND-GATE-OBJ-POWER-LEVEL(x) = 0.0

 form Make-NAND-GATE-Names-Unique
  unique-names-class('NAND-GATE-OBJ, true)
%-----------------------------------------------------------------
function NAND-GATE-OBJ-UPDATE1 (subsystem  : subsystem-obj,
                                nand-gate  : NAND-GATE-OBJ) =

 format(debug-on, "NAND-GATE-OBJ-UPDATE on ~s~%", name(nand-gate));

 let (in1 : boolean = get-import('in1, subsystem, nand-gate),
      in2 : boolean = get-import('in2, subsystem, nand-gate))

 set-export(subsystem, nand-gate, 'out1, ~(in1 & in2))
%-----------------------------------------------------------------
function NAND-GATE-OBJ-NEW-UPDATE (subsystem  : subsystem-obj,
                                   nand-gate  : NAND-GATE-OBJ) =

 format(t, "NAND-GATE-OBJ-NEW-UPDATE on ~s~%", name(nand-gate))
```

E.4 Nor-Gate

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  nor-gate.re

 var NOR-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var NOR-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}
```

```
 var NOR-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var NOR-GATE-OBJ-COEFFICIENTS : map(NOR-GATE-OBJ, set(name-value-obj))
  computed-using
    NOR-GATE-OBJ-COEFFICIENTS(x) = {}


 var NOR-GATE-OBJ-UPDATE-FUNCTION : map(NOR-GATE-OBJ, symbol)
   computed-using
     NOR-GATE-OBJ-UPDATE-FUNCTION(x) = 'NOR-GATE-OBJ-UPDATE1

% Other Attributes:
 var NOR-GATE-OBJ-DELAY : map(NOR-GATE-OBJ, integer)
   computed-using
    NOR-GATE-OBJ-DELAY(x) = 0

 var NOR-GATE-OBJ-MANUFACTURER :  map(NOR-GATE-OBJ, string)
   computed-using
    NOR-GATE-OBJ-MANUFACTURER(x) = " "

 var NOR-GATE-OBJ-MIL-SPEC? :  map(NOR-GATE-OBJ, boolean)
   computed-using
    NOR-GATE-OBJ-MIL-SPEC?(x) = nil

 var NOR-GATE-OBJ-POWER-LEVEL :  map(NOR-GATE-OBJ, real)
   computed-using
    NOR-GATE-OBJ-POWER-LEVEL(x) = 0.0

 form Make-NOR-GATE-Names-Unique
  unique-names-class('NOR-GATE-OBJ, true)
%-----------------------------------------------------------------
function NOR-GATE-OBJ-UPDATE1 (subsystem : subsystem-obj,
                               nor-gate  : NOR-GATE-OBJ) =


 format(debug-on, "NOR-GATE-OBJ-UPDATE on ~s~%", name(nor-gate));

 let (in1 : boolean = get-import('in1, subsystem, nor-gate),
      in2 : boolean = get-import('in2, subsystem, nor-gate))

 set-export(subsystem, nor-gate, 'out1, ~(in1 or in2))
%-----------------------------------------------------------------
function NOR-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                  nor-gate  : NOR-GATE-OBJ) =

 format(t, "NOR-GATE-OBJ-NEW-UPDATE on ~s~%", name(nor-gate))
```

E.5 Not-Gate

```
!! in-package("RU")
!! in-grammar('user)


%% File name: not-gate.re

 var NOT-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var NOT-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var NOT-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var NOT-GATE-OBJ-COEFFICIENTS : map(NOT-GATE-OBJ, set(name-value-obj))
  computed-using
    NOT-GATE-OBJ-COEFFICIENTS(x) = {}


 var NOT-GATE-OBJ-UPDATE-FUNCTION : map(NOT-GATE-OBJ, symbol)
   computed-using
     NOT-GATE-OBJ-UPDATE-FUNCTION(x) = 'NOT-GATE-OBJ-UPDATE1

% Other Attributes:
 var NOT-GATE-OBJ-DELAY : map(NOT-GATE-OBJ, integer)
   computed-using
    NOT-GATE-OBJ-DELAY(x) = 0

 var NOT-GATE-OBJ-MANUFACTURER :  map(NOT-GATE-OBJ, string)
   computed-using
    NOT-GATE-OBJ-MANUFACTURER(x) = " "

 var NOT-GATE-OBJ-MIL-SPEC? :  map(NOT-GATE-OBJ, boolean)
   computed-using
    NOT-GATE-OBJ-MIL-SPEC?(x) = nil

 var NOT-GATE-OBJ-POWER-LEVEL :  map(NOT-GATE-OBJ, real)
   computed-using
    NOT-GATE-OBJ-POWER-LEVEL(x) = 0.0

 form Make-NOT-GATE-Names-Unique
  unique-names-class('NOT-GATE-OBJ, true)
```

```
%-----------------------------------------------------------------------
function NOT-GATE-OBJ-UPDATE1 (subsystem : subsystem-obj,
                               not-gate  : NOT-GATE-OBJ) =

 format(debug-on, "NOT-GATE-OBJ-UPDATE on ~s~%", name(not-gate));

 let (in1 : boolean = get-import('in1, subsystem, not-gate))

 set-export(subsystem, not-gate, 'out1, ~(in1))
%-----------------------------------------------------------------------
function NOT-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                  not-gate  : NOT-GATE-OBJ) =

 format(t, "NOT-GATE-OBJ-NEW-UPDATE on ~s~%", name(not-gate))
```

E.6 JK-Flip-Flop

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  jk-flip-flop.re

 var JK-FLIP-FLOP-OBJ       : object-class subtype-of Primitive-Obj

 var JK-FLIP-FLOP-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'J,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),
       set-attrs (make-object('import-obj),
                   'import-name, 'K,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),
       set-attrs (make-object('import-obj),
                   'import-name, 'Clk,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var JK-FLIP-FLOP-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'Q,
                   'export-category, 'signal,
                   'export-type-data, 'boolean),
        set-attrs (make-object('export-obj),
                   'export-name, 'Q-Bar,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}
```

```
 var JK-FLIP-FLOP-OBJ-COEFFICIENTS : map(JK-FLIP-FLOP-OBJ, set(name-value-obj))
  computed-using
    JK-FLIP-FLOP-OBJ-COEFFICIENTS(x) = {}


 var JK-FLIP-FLOP-OBJ-UPDATE-FUNCTION : map(JK-FLIP-FLOP-OBJ, symbol)
   computed-using
     JK-FLIP-FLOP-OBJ-UPDATE-FUNCTION(x) = 'JK-FLIP-FLOP-OBJ-UPDATE1

% Other Attributes:
 var JK-FLIP-FLOP-OBJ-DELAY : map(JK-FLIP-FLOP-OBJ, integer)
   computed-using
    JK-FLIP-FLOP-OBJ-DELAY(x) = 0

 var JK-FLIP-FLOP-OBJ-MANUFACTURER :  map(JK-FLIP-FLOP-OBJ, string)
   computed-using
    JK-FLIP-FLOP-OBJ-MANUFACTURER(x) = " "

 var JK-FLIP-FLOP-OBJ-MIL-SPEC? :  map(JK-FLIP-FLOP-OBJ, boolean)
   computed-using
    JK-FLIP-FLOP-OBJ-MIL-SPEC?(x) = nil

 var JK-FLIP-FLOP-OBJ-POWER-LEVEL :  map(JK-FLIP-FLOP-OBJ, real)
   computed-using
    JK-FLIP-FLOP-OBJ-POWER-LEVEL(x) = 0.0

 var JK-FLIP-FLOP-OBJ-SET-UP-DELAY :  map(JK-FLIP-FLOP-OBJ, integer)
   computed-using
    JK-FLIP-FLOP-OBJ-SET-UP-DELAY(x) = 0

 var JK-FLIP-FLOP-OBJ-HOLD-DELAY :  map(JK-FLIP-FLOP-OBJ, real)
   computed-using
    JK-FLIP-FLOP-OBJ-HOLD-DELAY(x) = 0.0

 var JK-FLIP-FLOP-OBJ-STATE :  map(JK-FLIP-FLOP-OBJ, boolean)
   computed-using
    JK-FLIP-FLOP-OBJ-STATE(x) = nil



 form Make-JK-FLIP-FLOP-Names-Unique
  unique-names-class('JK-FLIP-FLOP-OBJ, true)

%-------------------------------------------------------------------------
function JK-FLIP-FLOP-OBJ-UPDATE1 (subsystem    : subsystem-obj,
                                   jk-flip-flop : JK-FLIP-FLOP-OBJ) =

 format(debug-on, "JK-FLIP-FLOP-OBJ-UPDATE on ~s~%", name(jk-flip-flop));

 let (j    : boolean = get-import('J, subsystem, jk-flip-flop),
```

```
      k   : boolean = get-import('K, subsystem, jk-flip-flop),
      clk : boolean = get-import('Clk, subsystem, jk-flip-flop))

 (if ~j & k & clk then
    JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop) <- nil

 elseif j & k & clk then
    JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop) <-
           ~JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop)
 elseif j & ~k & clk then
    JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop) <- true
 );
 set-export(subsystem, jk-flip-flop, 'Q,
         JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop));
 set-export(subsystem, jk-flip-flop, 'Q-Bar,
         ~JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop))
%-------------------------------------------------------------------------

function JK-FLIP-FLOP-OBJ-NEW-UPDATE (subsystem    : subsystem-obj,
                                      jk-flip-flop : JK-FLIP-FLOP-OBJ) =

 format(t, "JK-FLIP-FLOP-OBJ-NEW-UPDATE on ~s~%", name(jk-flip-flop))
```

E.7 switch

```
!! in-package("RU")
!! in-grammar('user)


%% File name:  switch.re

 var SWITCH-OBJ       : object-class subtype-of Primitive-Obj

 var SWITCH-OBJ-INPUT-DATA   : set(import-obj) =  {}

 var SWITCH-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var SWITCH-OBJ-COEFFICIENTS : map(SWITCH-OBJ, set(name-value-obj))
  computed-using
    SWITCH-OBJ-COEFFICIENTS(x) = {}


 var SWITCH-OBJ-UPDATE-FUNCTION : map(SWITCH-OBJ, symbol)
   computed-using
```

```
    SWITCH-OBJ-UPDATE-FUNCTION(x) = 'SWITCH-OBJ-UPDATE1

% Other Attributes:
 var SWITCH-OBJ-MANUFACTURER :  map(SWITCH-OBJ, string)
   computed-using
    SWITCH-OBJ-MANUFACTURER(x) = " "

 var SWITCH-OBJ-DEBOUNCED :  map(SWITCH-OBJ, boolean)
   computed-using
    SWITCH-OBJ-DEBOUNCED(x) = nil

 var SWITCH-OBJ-DELAY :  map(SWITCH-OBJ, integer)
   computed-using
    SWITCH-OBJ-DELAY(x) = 0

 var SWITCH-OBJ-POSITION :  map(SWITCH-OBJ, symbol)
   computed-using
    SWITCH-OBJ-POSITION(x) = 'on

 form Make-SWITCH-Names-Unique
  unique-names-class('SWITCH-OBJ, true)

%-----------------------------------------------------------------------
function SWITCH-OBJ-UPDATE1 (subsystem : subsystem-obj,
                             switch    : SWITCH-OBJ) =

 format(debug-on, "SWITCH-GATE-OBJ-UPDATE on ~s~%", name(switch));

 let (signal : boolean = nil)

 (if SWITCH-OBJ-POSITION(switch) = 'ON then
    signal <- true
 );

 set-export(subsystem, switch, 'out1, signal)
%-----------------------------------------------------------------------

function SWITCH-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                switch  : SWITCH-OBJ) =

 format(t, "SWITCH-OBJ-NEW-UPDATE on ~s~%", name(switch))
```

E.8 LED

```
!! in-package("RU")
!! in-grammar('user)
```

```
%% File name:  led.re

 var LED-OBJ       : object-class subtype-of Primitive-Obj

 var LED-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var LED-OBJ-OUTPUT-DATA  : set(export-obj) = {}

 var LED-OBJ-COEFFICIENTS : map(LED-OBJ, set(name-value-obj))
  computed-using
    LED-OBJ-COEFFICIENTS(x) = {}


 var LED-OBJ-UPDATE-FUNCTION : map(LED-OBJ, symbol)
   computed-using
     LED-OBJ-UPDATE-FUNCTION(x) = 'LED-OBJ-ON-OFF-UPDATE

% Other Attributes:
 var LED-OBJ-MANUFACTURER :  map(LED-OBJ, string)
   computed-using
    LED-OBJ-MANUFACTURER(x) = " "

 var LED-OBJ-COLOR :  map(LED-OBJ, symbol)
   computed-using
    LED-OBJ-COLOR(x) = 'red

 form Make-LED-Names-Unique
  unique-names-class('LED-OBJ, true)

%-----------------------------------------------------------------
function LED-OBJ-ON-OFF-UPDATE (subsystem : subsystem-obj,
                                led       : LED-OBJ) =

 format(debug-on, "LED-OBJ-ON-OFF-UPDATE on ~s~%", name(led));

 let (display-value : symbol = 'off)

 (if get-import('in1, subsystem, led) then
    display-value <- 'on
 );
 format(t, "LED ~s = ~s~%", name(led), display-value)
%-----------------------------------------------------------------

function LED-OBJ-T-F-UPDATE (subsystem : subsystem-obj,
                                 led  : LED-OBJ) =

 format(debug-on, "LED-OBJ-T-F-UPDATE on ~s~%", name(led));
```

```
let (display-value : symbol = 'false)

(if get-import('in1, subsystem, led) then
   display-value <- 'true
);
format(t, "LED ~s = ~s~%", name(led), display-value)
```

E.9 Counter

```
!! in-package("RU")
!! in-grammar('user)


%% File name:  counter.re

 var COUNTER-OBJ       : object-class subtype-of Primitive-Obj

 var COUNTER-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'clock,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'reset,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var COUNTER-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'lsb,
                   'export-category, 'signal,
                   'export-type-data, 'boolean),

        set-attrs (make-object('export-obj),
                   'export-name, 'msb,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}


 var COUNTER-OBJ-COEFFICIENTS : map(COUNTER-OBJ, set(name-value-obj))
  computed-using
    COUNTER-OBJ-COEFFICIENTS(x) =
      {set-attrs (make-object('name-value-obj), 'name-value-name, 'max-count,
                                                'name-value-value, 3)}
```

```
 var COUNTER-OBJ-UPDATE-FUNCTION : map(COUNTER-OBJ, symbol)
   computed-using
     COUNTER-OBJ-UPDATE-FUNCTION(x) = 'COUNTER-OBJ-UPDATE1

% Other Attributes:

 var COUNTER-OBJ-COUNT : map(COUNTER-OBJ, integer)
   computed-using
    COUNTER-OBJ-COUNT(x) = 0

 var COUNTER-OBJ-DELAY : map(COUNTER-OBJ, integer)
   computed-using
    COUNTER-OBJ-DELAY(x) = 0

 var COUNTER-OBJ-MANUFACTURER :  map(COUNTER-OBJ, string)
   computed-using
    COUNTER-OBJ-MANUFACTURER(x) = " "

 var COUNTER-OBJ-MIL-SPEC? :  map(COUNTER-OBJ, boolean)
   computed-using
    COUNTER-OBJ-MIL-SPEC?(x) = nil

 var COUNTER-OBJ-POWER-LEVEL :  map(COUNTER-OBJ, real)
   computed-using
    COUNTER-OBJ-POWER-LEVEL(x) = 0.0

 form Make-COUNTER-Names-Unique
  unique-names-class('COUNTER-OBJ, true)

%-------------------------------------------------------------------------
function COUNTER-OBJ-UPDATE1 (subsystem : subsystem-obj,
                              counter   : COUNTER-OBJ) =

 format(debug-on, "COUNTER-OBJ-UPDATE1 on ~s~%", name(counter));

 let (clock : boolean = get-import('clock, subsystem, counter),
      reset : boolean = get-import('reset, subsystem, counter))

 (if reset then
    COUNTER-OBJ-COUNT(counter) <- 0
  elseif clock then
    COUNTER-OBJ-COUNT(counter) <- COUNTER-OBJ-COUNT(counter) +1
 );
 (if COUNTER-OBJ-COUNT(counter) >
        get-coefficient-value(counter, 'max-count) then
    COUNTER-OBJ-COUNT(counter) <- 0
 );
  if COUNTER-OBJ-COUNT(counter) = 0 then
    set-export(subsystem, counter, 'msb, nil);
    set-export(subsystem, counter, 'lsb, nil)
  elseif COUNTER-OBJ-COUNT(counter) = 1 then
```

```
    set-export(subsystem, counter, 'msb, nil);
    set-export(subsystem, counter, 'lsb, true)
  elseif COUNTER-OBJ-COUNT(counter) = 2 then
    set-export(subsystem, counter, 'msb, true);
    set-export(subsystem, counter, 'lsb, nil)
  elseif COUNTER-OBJ-COUNT(counter) = 3 then
    set-export(subsystem, counter, 'msb, true);
    set-export(subsystem, counter, 'lsb, true)
```

E.10 Half-Adder

```
!! in-package("RU")
!! in-grammar('user)


%% File name:  half-adder.re

 var HALF-ADDER-OBJ       : object-class subtype-of Primitive-Obj

 var HALF-ADDER-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}


 var HALF-ADDER-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 's,
                   'export-category, 'signal,
                   'export-type-data, 'boolean),
        set-attrs (make-object('export-obj),
                   'export-name, 'c,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var HALF-ADDER-OBJ-COEFFICIENTS : map(HALF-ADDER-OBJ, set(name-value-obj))
  computed-using
    HALF-ADDER-OBJ-COEFFICIENTS(x) = {}


 var HALF-ADDER-OBJ-UPDATE-FUNCTION : map(HALF-ADDER-OBJ, symbol)
   computed-using
```

```
    HALF-ADDER-OBJ-UPDATE-FUNCTION(x) = 'HALF-ADDER-OBJ-UPDATE1

% Other Attributes:
 var HALF-ADDER-OBJ-DELAY : map(HALF-ADDER-OBJ, integer)
   computed-using
    HALF-ADDER-OBJ-DELAY(x) = 0

 var HALF-ADDER-OBJ-MANUFACTURER :  map(HALF-ADDER-OBJ, string)
   computed-using
    HALF-ADDER-OBJ-MANUFACTURER(x) = " "

 var HALF-ADDER-OBJ-MIL-SPEC? :  map(HALF-ADDER-OBJ, boolean)
   computed-using
    HALF-ADDER-OBJ-MIL-SPEC?(x) = nil

 var HALF-ADDER-OBJ-POWER-LEVEL :  map(HALF-ADDER-OBJ, real)
   computed-using
    HALF-ADDER-OBJ-POWER-LEVEL(x) = 0.0

 form Make-HALF-ADDER-Names-Unique
  unique-names-class('HALF-ADDER-OBJ, true)

%-------------------------------------------------------------------
function HALF-ADDER-OBJ-UPDATE1 (subsystem  : subsystem-obj,
                                 half-adder : HALF-ADDER-OBJ) =

 let (in1 : boolean = get-import('in1, subsystem, half-adder),
      in2 : boolean = get-import('in2, subsystem, half-adder))

 if ~in1 and ~in2 then
   set-export(subsystem, half-adder, 's, nil);
   set-export(subsystem, half-adder, 'c, nil)
 elseif in1 and ~in2 then
   set-export(subsystem, half-adder, 's, true);
   set-export(subsystem, half-adder, 'c, nil)
 elseif ~in1 and in2 then
   set-export(subsystem, half-adder, 's, true);
   set-export(subsystem, half-adder, 'c, nil)
 elseif in1 and in2 then
   set-export(subsystem, half-adder, 's, nil);
   set-export(subsystem, half-adder, 'c, true)
```

E.11 Decoder

```
!! in-package("RU")
!! in-grammar('user)
```

```
%% File name:  decoder.re

 var DECODER-OBJ        : object-class subtype-of Primitive-Obj

 var DECODER-OBJ-INPUT-DATA    : set(import-obj) =
        {set-attrs (make-object('import-obj),
                    'import-name, 'in1,
                    'import-category, 'signal,
                    'import-type-data, 'boolean),

         set-attrs (make-object('import-obj),
                    'import-name, 'in2,
                    'import-category, 'signal,
                    'import-type-data, 'boolean),

         set-attrs (make-object('import-obj),
                    'import-name, 'in3,
                    'import-category, 'signal,
                    'import-type-data, 'boolean)}

 var DECODER-OBJ-OUTPUT-DATA  : set(export-obj) =
        {set-attrs (make-object('export-obj),
                    'export-name, 'm0,
                    'export-category, 'signal,
                    'export-type-data, 'boolean),

         set-attrs (make-object('export-obj),
                    'export-name, 'm1,
                    'export-category, 'signal,
                    'export-type-data, 'boolean),

         set-attrs (make-object('export-obj),
                    'export-name, 'm2,
                    'export-category, 'signal,
                    'export-type-data, 'boolean),

         set-attrs (make-object('export-obj),
                    'export-name, 'm3,
                    'export-category, 'signal,
                    'export-type-data, 'boolean),

         set-attrs (make-object('export-obj),
                    'export-name, 'm4,
                    'export-category, 'signal,
                    'export-type-data, 'boolean),

         set-attrs (make-object('export-obj),
                    'export-name, 'm5,
                    'export-category, 'signal,
                    'export-type-data, 'boolean),
```

```
        set-attrs (make-object('export-obj),
                   'export-name, 'm6,
                   'export-category, 'signal,
                   'export-type-data, 'boolean),

        set-attrs (make-object('export-obj),
                   'export-name, 'm7,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

var DECODER-OBJ-COEFFICIENTS : map(DECODER-OBJ, set(name-value-obj))
 computed-using
   DECODER-OBJ-COEFFICIENTS(x) = {}


var DECODER-OBJ-UPDATE-FUNCTION : map(DECODER-OBJ, symbol)
  computed-using
    DECODER-OBJ-UPDATE-FUNCTION(x) = 'DECODER-OBJ-UPDATE1

% Other Attributes:
 var DECODER-OBJ-DELAY : map(DECODER-OBJ, integer)
   computed-using
    DECODER-OBJ-DELAY(x) = 0

 var DECODER-OBJ-MANUFACTURER :  map(DECODER-OBJ, string)
   computed-using
    DECODER-OBJ-MANUFACTURER(x) = " "

 var DECODER-OBJ-MIL-SPEC? :  map(DECODER-OBJ, boolean)
   computed-using
    DECODER-OBJ-MIL-SPEC?(x) = nil

 var DECODER-OBJ-POWER-LEVEL :  map(DECODER-OBJ, real)
   computed-using
    DECODER-OBJ-POWER-LEVEL(x) = 0.0

 form Make-DECODER-Names-Unique
  unique-names-class('DECODER-OBJ, true)

%-----------------------------------------------------------------------
function DECODER-OBJ-UPDATE1 (subsystem : subsystem-obj,
                              decoder   : DECODER-OBJ) =

 let (x : boolean = get-import('in1, subsystem, decoder),
      y : boolean = get-import('in2, subsystem, decoder),
      z : boolean = get-import('in3, subsystem, decoder))

%  set all outputs to false to start; don't want any left-over
%     values adversely affecting output.

 set-export(subsystem, decoder, 'm0, nil);
```

```
set-export(subsystem, decoder, 'm1, nil);
set-export(subsystem, decoder, 'm2, nil);
set-export(subsystem, decoder, 'm3, nil);
set-export(subsystem, decoder, 'm4, nil);
set-export(subsystem, decoder, 'm5, nil);
set-export(subsystem, decoder, 'm6, nil);
set-export(subsystem, decoder, 'm7, nil);

if ~x and ~y and ~z then
  set-export(subsystem, decoder, 'm0, true)
elseif ~x and ~y and z then
  set-export(subsystem, decoder, 'm1, true)
elseif ~x and y and ~z then
  set-export(subsystem, decoder, 'm2, true)
elseif ~x and y and z then
  set-export(subsystem, decoder, 'm3, true)
elseif x and ~y and ~z then
  set-export(subsystem, decoder, 'm4, true)
elseif x and ~y and z then
  set-export(subsystem, decoder, 'm5, true)
elseif x and y and ~z then
  set-export(subsystem, decoder, 'm6, true)
elseif x and y and z then
  set-export(subsystem, decoder, 'm7, true)
```

E.12 MUX

```
!! in-package("RU")
!! in-grammar('user)


%% File name: mux.re

 var MUX-OBJ       : object-class subtype-of Primitive-Obj

 var MUX-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in0,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
```

```
                    'import-type-data, 'boolean),

        set-attrs (make-object('import-obj),
                    'import-name, 'in3,
                    'import-category, 'signal,
                    'import-type-data, 'boolean),

        set-attrs (make-object('import-obj),
                    'import-name, 's0,
                    'import-category, 'signal,
                    'import-type-data, 'boolean),

        set-attrs (make-object('import-obj),
                    'import-name, 's1,
                    'import-category, 'signal,
                    'import-type-data, 'boolean)}

 var MUX-OBJ-OUTPUT-DATA  : set(export-obj) =
        {set-attrs (make-object('export-obj),
                    'export-name, 'out1,
                    'export-category, 'signal,
                    'export-type-data, 'boolean)}

 var MUX-OBJ-COEFFICIENTS : map(MUX-OBJ, set(name-value-obj))
  computed-using
    MUX-OBJ-COEFFICIENTS(x) = {}


 var MUX-OBJ-UPDATE-FUNCTION : map(MUX-OBJ, symbol)
   computed-using
     MUX-OBJ-UPDATE-FUNCTION(x) = 'MUX-OBJ-UPDATE1

% Other Attributes:
 var MUX-OBJ-DELAY : map(MUX-OBJ, integer)
   computed-using
    MUX-OBJ-DELAY(x) = 0

 var MUX-OBJ-MANUFACTURER :  map(MUX-OBJ, string)
   computed-using
    MUX-OBJ-MANUFACTURER(x) = " "

 var MUX-OBJ-MIL-SPEC? :  map(MUX-OBJ, boolean)
   computed-using
    MUX-OBJ-MIL-SPEC?(x) = nil

 var MUX-OBJ-POWER-LEVEL :  map(MUX-OBJ, real)
   computed-using
    MUX-OBJ-POWER-LEVEL(x) = 0.0

 form Make-MUX-Names-Unique
  unique-names-class('MUX-OBJ, true)
```

```
%-----------------------------------------------------------------------
function MUX-OBJ-UPDATE1 (subsystem : subsystem-obj,
                          mux       : MUX-OBJ) =

 let (s0 : boolean = get-import('s0, subsystem, mux),
      s1 : boolean = get-import('s1, subsystem, mux))

 if ~s0 and ~s1 then
   set-export(subsystem, mux, 'out1,
              get-import('in0, subsystem, mux))
 elseif s0 and ~s1 then
   set-export(subsystem, mux, 'out1,
              get-import('in1, subsystem, mux))
 elseif ~s0 and s1 then
   set-export(subsystem, mux, 'out1,
              get-import('in2, subsystem, mux))
 elseif s0 and s1 then
   set-export(subsystem, mux, 'out1,
              get-import('in3, subsystem, mux))
```

Vita

Captain Cynthia G. Anderson was born on 29 July 1955 in Burlington, Vermont and graduated as valedictorian from Mount Mansfield Union High School in Jericho Center, Vermont in June, 1972. She enlisted in the Air Force in April 1975 and completed technical training for computer programming at Sheppard AFB, Texas in September, 1975. After programming assignments in finance/budget and personnel/training functional areas at Peterson AFB, Colorado and Randolph AFB, Texas, she was accepted into the Airmen Education and Commissioning Program and entered the University of Oklahoma at Norman, Oklahoma in August, 1980. Upon graduating with special distinction and a Bachelor of Science in Computer Science degree in December 1982, she attended Officer Training School and received her commission as first honor graduate on 1 April 1983. As a new lieutenant, she was assigned to Tinker AFB, Oklahoma where she maintained the AWACS airborne operational computer program software and later was responsible for development test and evaluation of all new software versions. She was reassigned to Lowry AFB, Colorado in July 1988 and managed software configuration control for a world-wide satellite communications system. In May 1991, Captain Anderson entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio to pursue a Master of Science degree in Computer Systems.

Permanent address: 2800 Lamb Blvd #274
Las Vegas Nevada 89121

Bibliography

1. Arango, Guillermo. "Domain Analysis: Art Form to Engineering Discipline," *ACM*, 152–159 (January 1989).
2. Bailor, Paul D. and others. "Formalization and Visualization of Domain-Specific Software Architectures (submitted for publication)," (1992).
3. Baldo, James. *Reuse in Practice Workshop Summary*. Technical Report, Institute for Defense Analysis, April 1990 (AD-A226 895).
4. Barstow, David R. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, *11*:1321– 1326 (November 1985).
5. Batory, Don and Sean O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components*. Technical Report TR-91-22, Austin, Texas: University of Texas, January 1992.
6. Booch, Grady. *Software Engineering With Ada, Second Edition*. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc, 1987.
7. Booth, Guy R. *Implementation of an Object-Oriented Flight Simulator D.C. Electrical System on a Hypercube Architecture*. MS thesis, AFIT/GCE/ENG/91D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1991.
8. Breeding, Kenneth J. *Digital Design Fundamentals*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989.
9. Brooks, Frederick P., Jr. "No Silver Bullet – Essence and Accidents of Software Engineering," *IEEE Computer*, 10–19 (April 1987).
10. D'Ippolito, Richard and Kenneth Lee. "Modeling Software Systems by Domains." *Tenth Automating Software Design Workshop*. American Association for Artificial Intelligence, April 1992.
11. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89*. 256–265. New York, NY: Association of Computing Machinery, Inc., 1989.
12. D'Ippolito, Richard S. and Charles P. Plinta. "Software Development Using Models," *ACM Sigsoft Software Engineering Notes* (October 1989).
13. Fischer, Charles N. and Richard J. LeBlanc, Jr. *Crafting a Compiler with C*. Redwood City, CA: Benjamin/Cummings Publishing Company, Inc, 1991.
14. Frakes, W. B. "Representation Methods for Software Reuse." *Proceedings: TRI-Ada '89*. 500–516. New York, NY: Association of Computing Machinery, Inc., 1989.
15. Freeman, Peter, editor. *Tutorial: Software Reusability*. Washington, D.C.: Computer Society Press of the IEEE, 1987.
16. Greenspan, Sol J. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition*. PhD dissertation, University of Toronto, Toronto, Ontario, Canada, 1984.
17. Holibaugh, Robert. "Reuse: Where to Begin and Why." *Reuse in Practice Workshop Summary*, edited by James Baldo. 74–79. 1990.
18. Iscoe, Neil. "Domain Modeling – Evolving Research." *Proceedings of the Sixth Annual Knowldege-Based Software Engineering Conference*. 300 – 304. 1991.

19. Iscoe, Neil Allen. *Domain-Specific Programming: An Object-Oriented and Knowledge-based Approach to Specification and Generation*. PhD dissertation, The University of Texas at Austin, Austin Texas, 1990.

20. Kang, Kyo C. and others. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990 (AD-A235 785).

21. Korth, Henry F. and Abraham Silberschatz. *Database System Concepts, 2nd edition*. New York, NY: McGraw-Hill, Inc., 1991.

22. Lane, Thomas G. *Studying Software Architectures through Design Spaces and Rules*. Technical Report CMU/SEI-90-TR-18, Software Engineering Institute, November 1990.

23. Lee, Kenneth J. and others. *An OOD Paradigm for Flight Simulators, Second Edition*. Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, September 1988 (AD-A204 849).

24. Lee, Kenneth J. and others. *Model-Based Software Development (Draft)*. Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.

25. Lockheed Software Technology Center. *Software User's Manual for the Automatic Programming Technologies For Avionics Software (APTAS) System*. Technical Report, Palo Alto, CA: Lockheed Software Technology Center, June 1991.

26. Lowry, Michael R. "Software Engineering in the Twenty-first Century." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. 627–654. Menlo Park, CA: AAAI Press/MIT Press, 1991.

27. Mano, M. Morris. *Computer System Architecture*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1976.

28. Neighbors, James M. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, *10*:564–574 (September 1984).

29. Perry, J. M. and M. Shaw. "The Role of Domain Independence in Promoting Software Reuse, Architectural Analysis of Systems." *Reuse in Practice Workshop Summary*, edited by James Baldo. 123–128. 1990.

30. Peterson, A. Spencer. "Coming to Terms with Software Reuse: A Model-based Approach," *ACM SIGSOFT Software Engineering Notes*, *16*:45–51 (April 1991).

31. Prieto-Díaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*:47–54 (April 1990).

32. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Proceedings of the 11th Annual International Computer Software and Application Conference*. 23–29. IEEE Computer Society Press, 1990.

33. Randour, Captain Mary Anne. *Creating and Manipulating a Domain Specific Formal Object Base*. MS thesis, AFIT/GCS/ENG/92D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

34. Reasoning Systems, Inc. *DIALECT User's Guide*. Palo Alto, CA, July 1990.

35. Reasoning Systems, Inc. *REFINE User's Guide*. Palo Alto, CA, May 1990.

36. Royce, Walker. "Reliable, Reusable Ada Components for Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)." *Proceedings: TRI-Ada '89*. 500–516. New York, NY: Association of Computing Machinery, Inc., 1989.

37. Ruegsegger, Ted. "Making Reuse Pay: The SIDPERS-3 RAPID Center," *IEEE Communications Magazine, 26, No. 8*:16–24 (Aug 1988).

38. Shaw, Mary. "Larger Scale Systems Require Higher-Level Abstractions," *ACM Sigsoft Software Engineering Notes, 14, No. 3*:143–146 (May 1988).

39. Smith, Douglas R. "KIDS – A Knowledge-Based Software Development System." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. 483–514. Menlo Park, CA: AAAI Press/MIT Press, 1991.

40. Spicer, Kelly L. *Mapping an Object-Oriented Requirements Analysis to a Design Architecture that Supports Reuse*. MS thesis, AFIT/GCS/ENG/90D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1990.

41. Stark, Michael E. and Eric W. Booth. "Using Ada to Maximize Verbatim Software Reuse." *Proceedings: TRI-Ada '89*. 278–290. New York, NY: Association of Computing Machinery, Inc., 1989.

42. Stewart, Jeff. "Software Architectures," *briefing* (August 1992).

43. Tracz, Will. "Summary of Implementation Working Group." *Reuse in Practice Workshop Summary*, edited by James Baldo. 10–19. 1990.

44. Weide, Lieutenant Timothy. *Visualization and Manipulation of a Formal Object Base (Draft)*. MS thesis, AFIT/GCS/ENG/93M, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
CREATING AND MANIPULATING FORMALIZED SOFTWARE ARCHITECTURES TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
Cynthia G. Anderson, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/92D-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ASC/RWWW
Wright-Patterson AFB, OH 45433-6583

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research investigated technology which enables sophisticated users to specify, generate, and maintain application software in domain-oriented terms. To realize this new technology, a development environment, called Architect, was designed and implemented. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. Architect depends upon the existence of a formal object base (or domain model) which was investigated by another student in related research. The research described in this thesis relied on the concept of a software architecture, which was a key to Architect's successful implementation. Various software architectures were evaluated and the Object-Connection-Update (OCU) model, developed by the Software Engineering Institute, was selected. The Software Refinery environment was used to implement the composition process which encompasses connecting specified domain objects into a composed application, performing semantic analysis on the composed application, and, if no errors are discovered, simulating the execution of the application. Architect was validated using both artificial and realistic domains and was found to be a solid foundation upon which to build a full-scale application composition system.

14. SUBJECT TERMS
computers, computer programs, software engineering, specifications, software architecture models, application composition systems, domain-modeling, domain-specific languagues

15. NUMBER OF PAGES
254

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL